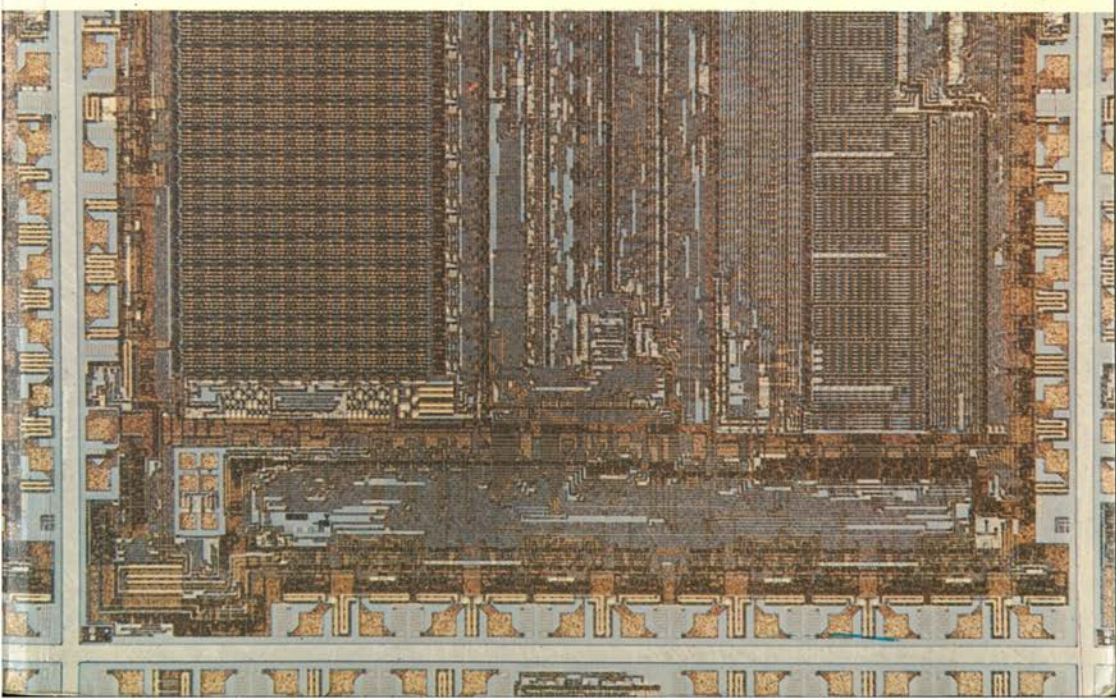
Programmazione dello Z80 e progettazione logica

EDIZIONE ITALIANA

A. OSBORNE
J. KANE
R. RECTOR
S.JACOBSON

GRUPPO EDITORIALE JACKSON

Programmazione dello Z80 e progettazione logica

di
Adam Osborne
Jerry Kane
Russell Rector
e
Susanna Jacobson

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

SOMMARIO

SOMMARIO (continua)

SOMMARIO (continua)

SOMMARIO (continua)

Capitolo **Pagina**

SOMMARIO (continua)

Capitolo **Pagina**

SOMMARIO (continua)

SOMMARIO (continua)

SOMMARIO (continua)

Capitolo | **Pagina**

SOMMARIO (continua)

SOMMARIO (continua)

Capitolo **Pagina**

SOMMARIO (continued)

Capitolo **Pagina**

INDICE DELLE FIGURE

INDICE DELLE TABELLE

INDICE ANALITICO

INDICE ANALITICO (continua)

Indice **Pagina**

INDICE ANALITICO (continua)

INDICE ANALITICO (continua)

Capitolo 1

INTRODUZIONE

LOGICA COMBINATORIA

Questo libro spiega come un programma in linguaggio assembly in un sistema a microcalcolatore possa sostituire una logica combinatoria — cioè l'uso combinato di dispositivi logici non programmabili, come quelli della serie logica digitale 7400 standard.

Se voi siete progettisti logici, questo libro v'insegnerà come fare il vostro lavoro in un modo nuovo — creando programmi in linguaggio assembly in un sistema a microcalcolatore.

Se voi siete programmatori, questo libro vi mostrerà come la programmazione abbia trovato un nuovo scopo — nel progetto logico.

Questo è un libro su "come fare ciò"; come tale, deve essere molto specifico, quindi fa riferimento diretto ad un particolare microcalcolatore, lo Z80.

Le compagnie che producono questo microcalcolatore sono:

ZILOG, INCORPORATED
10460 Bubb Road
Cupertino, California 95014

MOSTEK, INCORPORATED
1215 West Crosby Road
Carrollton, Texas 75006

CIO' CHE QUESTO LIBRO SUPPONE VOI CONOSCIATE

Questo libro è un seguito a An Introduction to Microcomputers, che era costituito da un volume singolo nella sua prima edizione, ma che è formato da due volumi nella sua seconda edizione.

An Introduction to Microcomputers descrive concettualmente i microprocessori e i microcalcolatori; non è indirizzato al modo pratico di implementare un concetto. Questo libro è orientato al modo pratico di implementazione.

Poichè questo libro costituisce un seguito, esso fa una sola ipotesi — cioè che voi abbiate letto, o conosciuto altrimenti, la materia contenuta in An Introduction to Microcomputers. Tuttavia, prima di lanciarvi in un progetto reale, avrete bisogno di letteratura commerciale che descriva specificatamente i dispositivi che avete scelto di usare.

Noterete, in particolare, che in questo libro non si descrivono nè hardware nè temporizzazioni, sia per la CPU dello Z80 che di ogni altro dispositivo del microcalcolatore; informazioni sufficienti si possono trovare in An Introduction to Microcomputers, Volume II - Some Real Products.

L'insieme delle istruzioni dello Z80 è descritto nel Capitolo 6, poichè questo libro tratta solo di programmazione.

COMPRENSIONE DEL LINGUAGGIO ASSEMBLY

Le istruzioni in linguaggio assembly sono le funzioni di trasferimento di un sistema a microcalcolatore; prese insieme, esse costituiscono un "insieme di istruzioni", che descrive le singole operazioni che un microcalcolatore può eseguire.

Si definiscono gli eventi che devono accadere serialmente in un sistema a microcalcolatore — come una sequenza di istruzioni che, prese insieme, costituiscono un programma in linguaggio assembly.

In realtà, la comprensione di ciò che le singole istruzioni fanno in un sistema a microcalcolatore è immediata; questo è uno degli aspetti più semplici del lavorare con microcalcolatori. Tuttavia ciò terrorizza indebitamente gli utilizzatori che sono nuovi alla programmazione. Se ciò vi riguarda, un consiglio — dimenticatevi della mnemonica e dell'insieme delle istruzioni; prendete le istruzioni una per volta così come le incontrerete in questo libro. Quando non capite ciò che fa un'istruzione, guardatelo nel Capitolo 6.

Lo "spettro" della "programmazione" vi ossessionerà solo se voi lo permetterete.

COME E' STATO STAMPATO QUESTO LIBRO

Avrete notato che il testo in questo libro è stato stampato in neretto ed in chiaro. **Ciò è stato fatto per aiutarvi a saltare quelle parti del libro che coprono materia il cui argomento vi è familiare. Voi potrete essere sicuri che le scritte in chiaro sviluppano un'informazione presentata precedentemente in neretto.** Quindi leggete solo le scritte in neretto, finchè non troverete un argomento che voi volete conoscere maggiormente; a questo punto cominciate a leggere le scritte in chiaro.

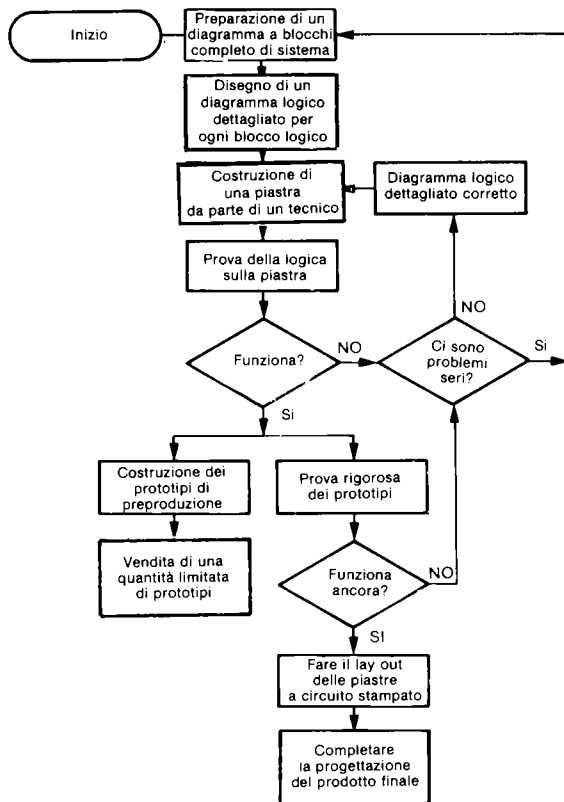
Capitolo 2

LINGUAGGIO ASSEMBLY E LOGICA DIGITALE

IL CICLO DI PROGETTAZIONE

CICLO DI UN PROGETTO LOGICO DIGITALE

Un prodotto che deve essere costruito con componenti discreti a logica digitale passerà attraverso un ben definito ciclo di progettazione.

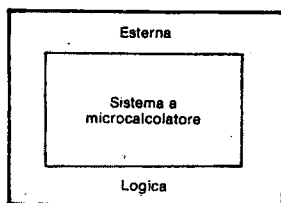
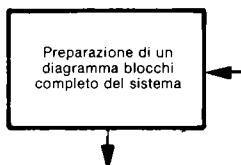
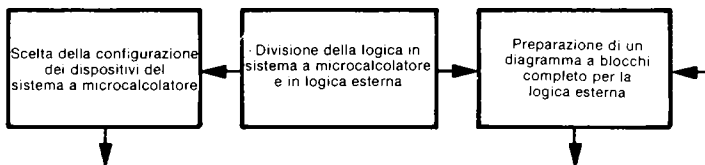
Supponiamo che si sia definito un prodotto — da un punto di vista di gestione del marketing.

Vi sono state presentate le specifiche di un prodotto che si identificano necessariamente con le caratteristiche e le prestazioni del prodotto; il vostro lavoro sarà quello di consegnare un progetto funzionante da produrre. **Il ciclo di progettazione procederà come rappresentato nella figura di pagina 2-1.**

In ogni ciclo di un progetto logico c'è un anello (loop) iterativo costoso e lento; come è stato illustrato sopra, esso consiste nei seguenti passi:

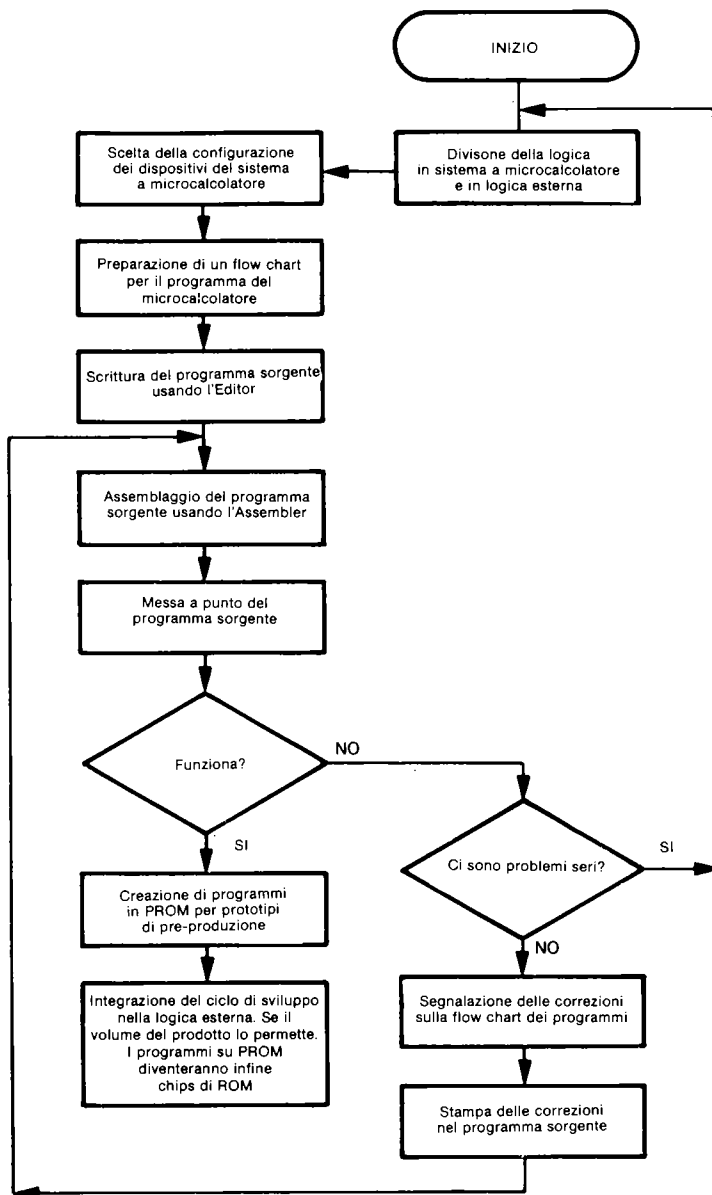
- Ridisegnare la logica
- Costruire una nuova piastra
- Provare la piastra per errori logici, tecnici o di componenti

Questo loop iterativo rende il progetto della logica combinatoria lento e costoso — non solo durante la fase iniziale del progetto, ma anche, e maggiormente, quando decidete in seguito di modificare o di aumentare il prodotto.

CICLO DI UN PROGETTO LOGICO CON MICROCALCOLATORE

Che cosa succede quando cominciate ad usare microcalcolatori? Anzitutto, una parte della vostra logica svanisce in una "scatola nera" — che è il sistema del microcalcolatore:

Il vostro primo passo:

ora deve essere espanso come segue:

Il frazionare la vostra applicazione in un sistema a microcalcolatore e in una logica digitale esterna, potrà sembrare una proposta difficile — se non comprendete che cosa può fare un sistema a microcalcolatore.

Infatti, **una volta che avete un microcalcolatore nel vostro prodotto, l'economia del sistema ne è favorita irresistibilmente facendo assumere alla "scatola nera" il maggior numero di compiti possibile; voi dovrete giustificare l'esistenza di ogni singola porta logica esterna.**

E' da ricordare che la memoria viene aggiunta con incrementi definiti. Al fine di espandere la logica implementata in un sistema a microcalcolatore, voi potete semplicemente scrivere sequenze d'istruzioni addizionali che risiederanno in memoria, che diversamente sarebbe sprecata; è perciò che l'incremento della memoria di programma costa molto poco.

Inoltre, confrontato col costo di sviluppo della logica digitale, lo sviluppo della logica a microcalcolatore è veloce e non costoso. **Un tipico ciclo di sviluppo di sistema a microcalcolatore può essere illustrato come nella figura di pagina precedente.**

Nel ciclo di sviluppo del microcalcolatore illustrato sopra ci sono ancora loop iterativi, ma, confrontato con lo sviluppo di logica digitale, il ciclo di sviluppo a microcalcolatore è associato a loop iterativi aventi tempo e costo minori.

Ogni microcalcolatore è supportato da un sistema di sviluppo. Le caratteristiche e il funzionamento di questi sistemi di sviluppo variano notevolmente da una compagnia all'altra; **tuttavia essi hanno queste capacità:**

1) Potete **simulare il sistema a microcalcolatore** che avete configurato senza creare necessariamente una piastra.

PROGRAMMA SORGENTE

2) Potere far eseguire un programma editor residente per **creare il vostro programma sorgente. Ricordiamo che ci si riferisce ad una sequenza di istruzioni in linguaggio assembly come ad un "Programma Sorgente".**

PROGRAMMA OGGETTO

3) Potete **assemblare un programma sorgente** proprio col sistema di sviluppo per creare un programma oggetto. E' da ricordare che il programma sorgente diventa una sequenza di bit (riferita ad un programma oggetto) prima che esso possa venire eseguito.

4) Potete **con certe condizioni eseguire il programma oggetto** per essere sicuri che esso funzioni.

Usando un tipico sistema di sviluppo di un microcalcolatore, potete passare attraverso parecchi cicli di sviluppo in più in un solo giorno, quando ogni ciclo di sviluppo in una implementazione totalmente a logica digitale avrebbe occupato una o due settimane. In un singolo ciclo di sviluppo potete fare molte correzioni di programma; in meno di un minuto potete fare una semplice correzione, equivalente ad aggiungere o togliere una porta (gate) (o una funzione MSI) da una piastra di logica digitale.

SIMULAZIONE DELLA LOGICA DIGITALE

Si è visto che la logica deve essere separata in logica interna ad un sistema a microcalcolatore e in logica esterna al sistema a microcalcolatore.

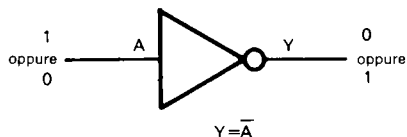
Ci interesseremo di due aspetti di questa separazione logica:

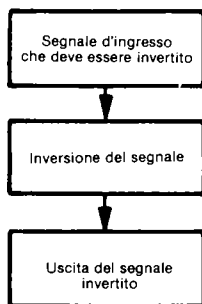
1) **Dobbiamo sviluppare alcuni criteri semplici per stimare ciò che un sistema a microcalcolatore può o non può fare,** criteri basati sulla capacità del linguaggio assembly di simulare logica digitale.

2) **Dobbiamo creare un programma per implementare le funzioni logiche che sono state assegnate al sistema a microcalcolatore.** Sfortunatamente, ci sono innumerevoli modi di scrivere un programma per un microcalcolatore. Una volta che si conosce a fondo il concetto di usare istruzioni per pilotare un sistema a microcalcolatore, **il passo successivo è imparare come scrivere programmi efficienti.**

Cominceremo descrivendo una semplice simulazione di logica digitale. Ciò è necessario come inizio perchè ci sono alcune differenze concettuali fondamentali tra logica digitale e logica programmata su microcalcolatore.

SIMULAZIONE CON MICROCALCOLATORE DI UN INVERTITORE DI SEGNALE

Supponiamo che si voglia invertire un singolo segnale

FLOW CHART

Nell'interesse di sviluppare delle buone abitudini dall'inizio, illustreremo l'invertitore di segnale con il seguente flowchart logico:

Sebbene non si userà mai un microcalcolatore per sostituire un invertitore di segnale, vale ancora la pena di esaminare come si potrebbe fare ciò.

UNA SEQUENZA DI EVENTI DEL MICROCALCOLATORE

REGISTRI DELLA CPU

Ricordiamo che il microcalcolatore Z80 ha i seguenti registri nella CPU:

Registri		Descrizione	Registri alternativi	
	F	Parole di stato del programma		F'
	A	Accumulatori primari		A'
B	C	Accumulatori secondari/Contatori di dati	B'	C'
D	E	Accumulatori secondari/Contatori di dati	D'	E'
H	L	Accumulatori secondari/Contatori di dati	H'	L'
SP		Stack Pointer		
PC		Program counter		
IX		Registro indice X		
IY		Registro indice Y		
	IV	Vettore d'interruzione		
	R	Contatore di rinfresco della memoria		

La singola istruzione:

```
CPL            ; Complementa l'accumulatore
```

DATO IN BIT

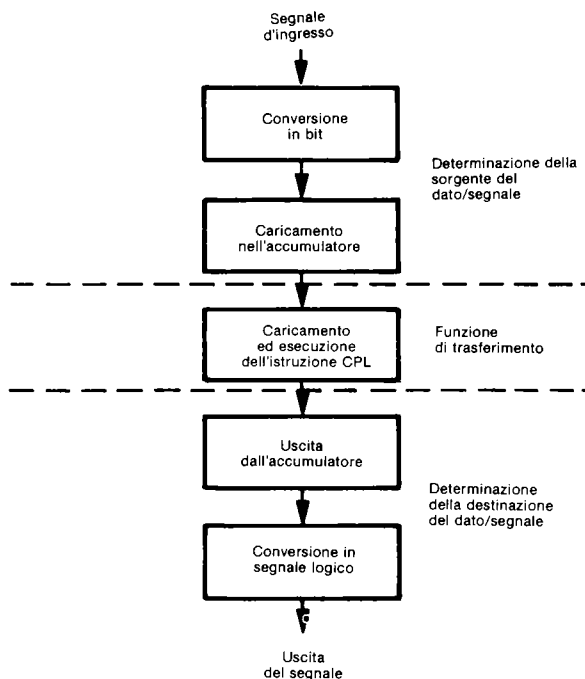
Quando è convertita in codice oggetto ed eseguita, inverte tutti gli otto bit dell'Accumulatore primario. Ma ciò non duplica l'invertitore. Anzitutto si deve scegliere un solo bit dell'Accumulatore per rappresentare il segnale che deve essere invertito. Ma quale?

DESTINAZIONE E SORGENTE DEL DATO

TEMPORIZZAZIONE DEL PROGRAMMA

Avendo deciso quale bit, come fa esso a raggiungere l'Accumulatore in prima posizione? E, una volta invertito, come fa il bit invertito a diventare nuovamente un segnale?

Se il codice oggetto dell'istruzione CPL deve essere eseguito per realizzare la inversione reale, come e quando il codice oggetto giungerà alla CPU? Chiaramente, l'esecuzione di questa istruzione deve avvenire dopo che il bit che deve essere invertito ha raggiunto l'Accumulatore.

I passi necessari per implementare un invertitore usando un microcalcolatore possono essere illustrati sviluppando il seguente flowchart:

Nella illustrazione precedente, fate molta attenzione alla divisione del problema in queste tre fasi:

1) **Determinazione della sorgente del dato/segnale.** Identifichiamo il dato che deve essere elaborato. Questo dato è trasferito in una locazione che può essere raggiunta dalla CPU (Central Processing Unit) del microcalcolatore.

2) **Esecuzione della funzione di trasferimento.** L'operazione reale che deve essere effettuata sul dato della sorgente sarà richiamata come"funzione di trasferimento".

3) **Determinazione della destinazione del dato/segnale**. I dati o i segnali sottoposti alla funzione di trasferimento, devono essere trasferiti a qualche destinazione.

Genereremo ora una sequenza di istruzioni per implementare le tre fasi della simulazione dell'invertitore illustrata sopra.

IMPLEMENTAZIONE DELLA FUNZIONE DI TRASFERIMENTO

DATO IN BIT

L'istruzione CPL inverte ogni bit dell'accumulatore.

L'istruzione CPL, quindi non specifica quale bit dell'Accumulatore rappresenta il segnale da invertire. Questa specificazione è implicita nel modo in cui il dato si presenta in ingresso e in uscita del sistema a microcalcolatore.

DETERMINAZIONE DELLE SORGENTI E DELLE DESTINAZIONI DEI DATI

Come fa il dato dell'Accumulatore a entrare e uscire dal sistema a microcalcolatore? Per rispondere a questa domanda, tratteremo una delle fondamentali forze (e complessità) dei microcalcolatori — la loro flessibilità.

LOGICA ESTERNA COME SORGENTE O DESTINAZIONE

INPUT/OUTPUT

Il segnale d'ingresso e il segnale d'uscita invertito sono proprio quali i loro nomi implicano — essi sono segnali. Ma, per il sistema a microcalcolatore essi sono "logica esterna". I trasferimenti d'informazioni tra la logica esterna e il sistema a microcalcolatore sono generalmente richiamati come Input/Output (o I/O). Durante una operazione di I/O, ricordiamo che il microcomputer è master e la logica esterna è slave. Ciò significa che il microcalcolatore deve indicare la direzione dell'operazione di I/O (input o output), e deve identificare la logica esterna a cui si deve accedere.

I/O IN UNA ZONA DI INDIRIZZO DI MEMORIA

La logica esterna deve decodificare uno specifico indirizzo di memoria come uno strobe di abilitazione, così che l'I/O è trattato come se si scrivesse o leggesse in memoria. **Supponiamo che la label INVD sia usata nel programma sorgente in linguaggio assembly per identificare il segnale che deve essere invertito. Questa è la sequenza di istruzioni che riprodurrà l'invertitore di segnale:**

```
LD   A,(INVD)     ; Carica l'Accumulatore da INVD
CPL               ; Complementa l'Accumulatore
LD   (INVD),A     ; Memorizza il contenuto dell'Accumulatore in INVD
```

In termini di dispositivi di microcalcolatore, la Figura 2-1 mostra la configurazione di microcalcolatore implicata.

Quando si esegue l'istruzione LD A,(INVD), la "Logica di Decodifica dell'Indirizzo" fa sì che la "Logica di Selezione" trasmetta il segnale "Dato In"al Bus dei Dati.

Ci sono otto linee nel Bus dei Dati; il numero della linea a cui il segnale del "Dato In" è collegato diventa il numero del bit significativo nell'Accumulatore. Quando l'istruzione LD A,(INVD) è stata eseguita, il contenuto del Bus dei Dati sarà nello Accumulatore.

Successivamente, si esegue l'istruzione CPL. Questa istruzione fa sì che ogni bit dell'Accumulatore sia complementato.

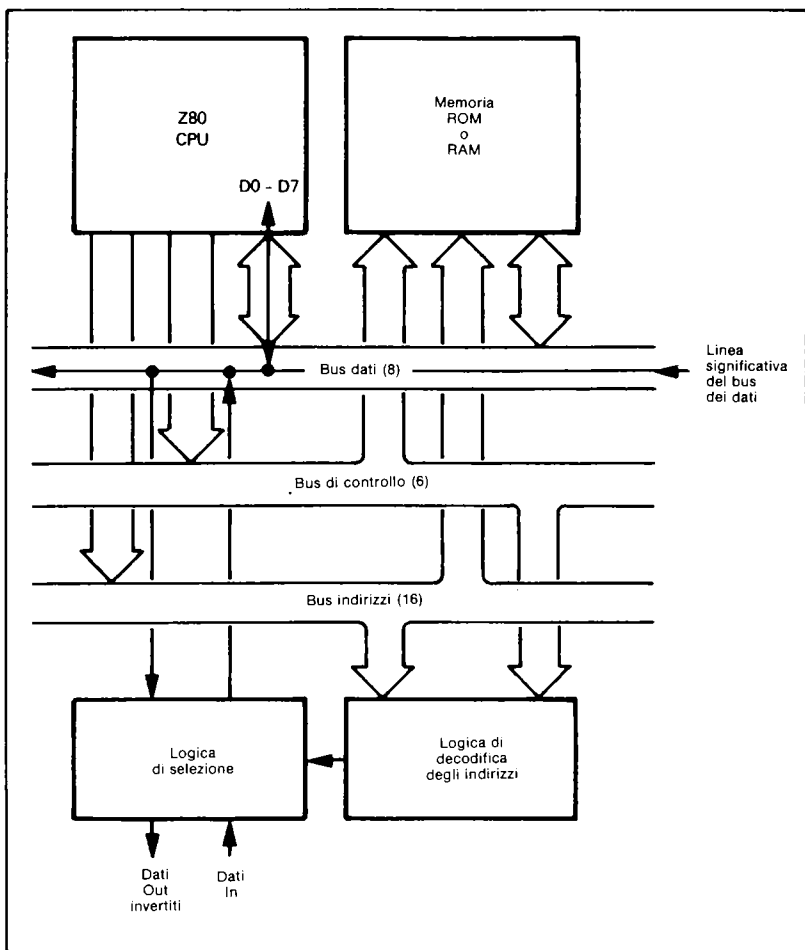



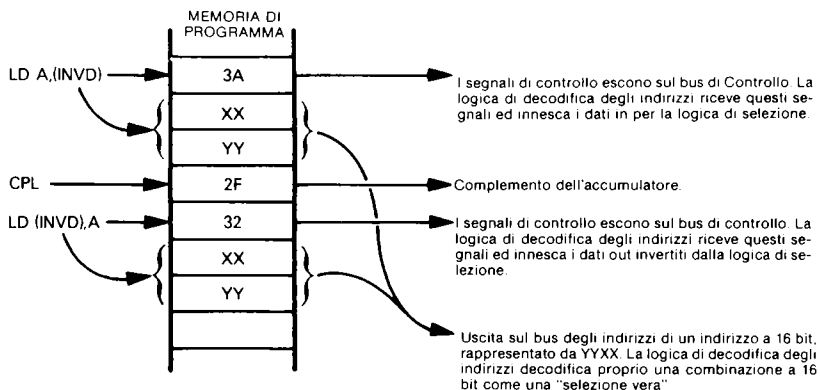
Figura 2-1. Configurazione per indirizzamento di I/O visto come Mappa di Memoria (Memory-Mapped)

Quando si esegue l'istruzione LD (INVD),A, il contenuto dell'Accumulatore è in uscita sul Bus dei Dati. La "Logica di Decodifica dell'Indirizzo" allora fa sì che la "Logica di Selezione" faccia uscire il contenuto di una singola linea del Bus dei Dati — che diventa il segnale invertito "Dato Out".

Poichè la "Logica di Selezione" ha i segnali "Dato In" e "Dato Out" collegati sulla stessa linea del Bus dei Dati, il "Dato Out" è il complemento del "Dato In" e l'invertitore di segnale è stato simulato.

Nel sistema a microcalcolatore devono essere presenti memorie ROM o RAM, poichè i codici oggetto delle tre istruzioni devono essere memorizzati nella memoria e prelevati (fetched) dalla memoria.

INTERPRETAZIONE DEL CODICE OGGETTO

Consideriamo in dettaglio il codice oggetto. Le tre istruzioni del programma sorgente diventano codici oggetto come segue:

Gli indirizzi dei byte della memoria di programmi entro i quali i codici oggetto sono memorizzati non sono importanti. Tuttavia, nessun byte di memoria, ROM o RAM, può avere l'indirizzo rappresentato da YYXX, poichè questo indirizzo seleziona la logica esterna.

Osservate che i due byte dell'indirizzo a 16 bit YYXX sono scambiati quando vengono memorizzati in memoria. Non c'è nulla di significativo riguardo a questa inversione, è proprio il modo in cui i dispositivi dello Z80 sono stati progettati.

I/O PER MEZZO DI PORTE DI I/O

Supponiamo ora che la comunicazione con la logica esterna avvenga per mezzo di un dispositivo d'interfaccia periferico di I/O.

Nelle istruzioni del programma sorgente in linguaggio assembly, la label INVD identificherà ora una porta di I/O. Questa è la sequenza di istruzione che riproduce l'invertitore di segnale:

```
IN    A,(INV)      ; Ingresso nell'Accumulatore dalla porta INVD
CPL                ; Complementa l'Accumulatore
OUT   (INVD),A     ; Uscita dell'Accumulatore nella porta INVD
```

In termini di hardware, la Figura 2-2 mostra la configurazione del microcalcolatore implicata.

Tutto ciò che abbiamo fatto aggiungendo il dispositivo Parallelo di I/O dello Z80 (Z80 PIO), è di fornire la "Decodifica dell'Indirizzo" e la "Logica di Selezione" necessaria ai segnali "Dati In" e "Dati Out" invertito. Ora il particolare bit, che è significativo, sarà individuato dal pin dello Z80 PIO al quale i segnali "Dati In" e "Dati Out" invertito sono collegati. A loro volta, questi pins saranno determinati dal modo in cui lo Z80 PIO è usato.

Il fatto che diverse opzioni siano disponibili nell'usare lo Z80 PIO non ha alcuna conseguenza immediata, in quanto ciò confonderebbe la vostra iniziale comprensione per ciò che riguarda la programmazione in linguaggio assembly. Noi, quindi, ignoreremo le istruzioni di controllo dei modi dello Z80 PIO, e supporremo semplicemente che si sia scelto l'appropriato controllo dei modi.

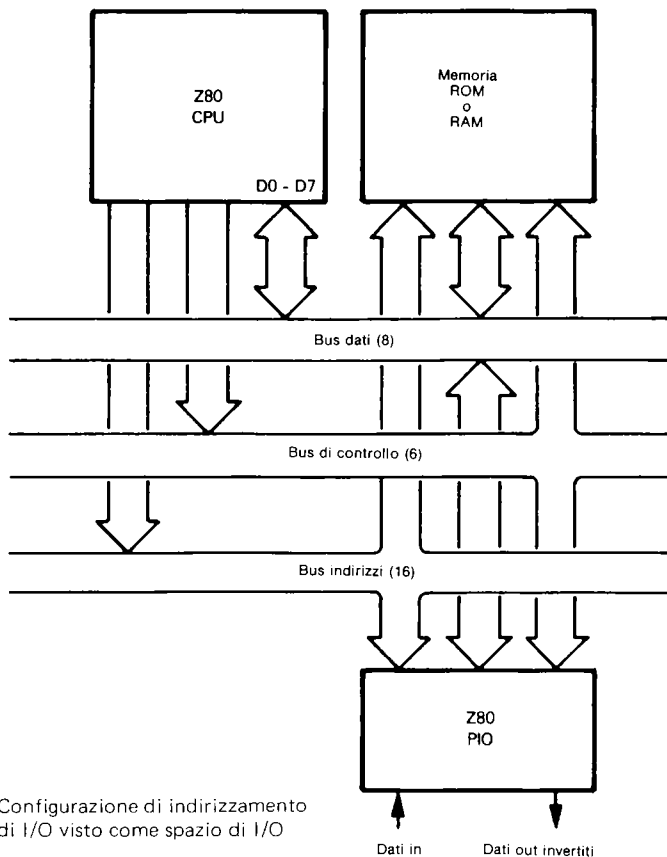



Figura 2-2. Configurazione di indirizzamento di I/O visto come spazio di I/O

INTERPRETAZIONE DEL CODICE OGGETTO

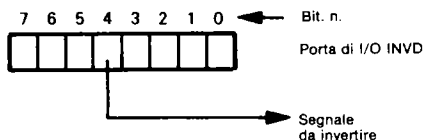
In questo caso, il codice oggetto per le tre istruzioni è interpretato come segue:

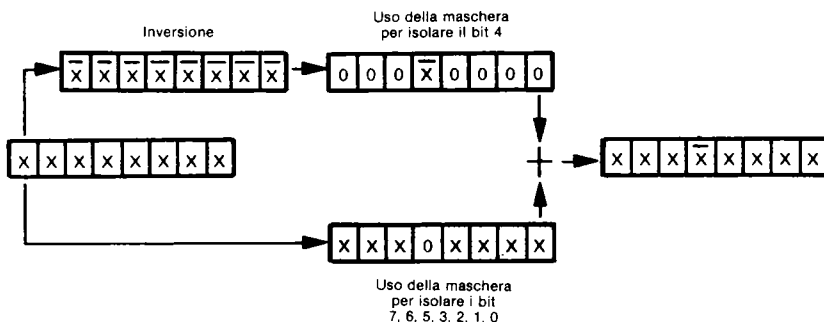
Istruzione	MEMORIA DI PROGRAMMA	
IN A,(INVD)	DB	Lo Z80 PIO decodifica segnali di controllo dalla CPU ed innesca dati in
	PP	Numero della porta di I/O, che esce sulle 8 linee minori del bus degli indirizzi
CPL	2F	Complemento dell'accumulatore
OUT (INVD),A	D3	Lo Z80 PIO decodifica segnali di controllo dalla CPU ed innesca dati out invertiti
	PP	Numero della porta di I/O, che esce sulle 8 linee minori del bus degli indirizzi

Ancora una volta, gli indirizzi dei byte di memoria programma entro i quali i codici oggetto precedenti sono memorizzati non saranno importanti.

SCELTA DEI PIN DELLA PORTA DI I/O

Osserviamo che stiamo complementando ogni bit della porta di I/O INVD, anche se solo un bit corrisponde al segnale che deve essere invertito.

Supponiamo che solo il pin 4 deve essere invertito:

MASCHERATURA DEL BIT

Possiamo usare una tecnica nota come "mascheratura" per invertire un singolo pin di una porta di I/O, lasciando stare tutti gli altri pin. Per esempio, la mascheratura può essere illustrata come segue:

Nella illustrazione precedente, X rappresenta un qualsiasi bit; $\overline{X}$ rappresenta il suo complemento.

La seguente sequenza di istruzioni invertirà il pin 4, lasciando gli altri pin come erano:

```
IN   A,(INVD)     ; Ingresso nell'Accumulatore dalla porta di I/O INVD
CPL               ; Complementa l'Accumulatore
AND  10H          ; Isolamento del bit 4
LD   B,A          ; Salvataggio nel registro B
IN   A,(INVD)     ; Ingresso nell'Accumulatore dalla porta di I/O INVD
AND  0EFH         ; Azzera il bit 4
OR   B            ; OR di A con B
OUT  (INVD),A     ; Uscita dell'Accumulatore nella porta di I/O INVD
```

H NEL CAMPO DELL'OPERANDO

H, come ultimo carattere nel campo dell'operando, specifica un valore di un dato esadecimale, immediato. In tale modo 0EFH rappresenta il valore binario:

ZERO COME GUIDA

I numeri esadecimali comincianti con i caratteri compresi tra A ed F sono preceduti da uno 0 per impedire all'assembler di confondere i numeri dai nomi delle variabili.

In termini di contenuto di registri, ciò accade quando la sequenza di istruzioni precedente viene eseguita (X rappresenta ancora un bit qualsiasi):

		Porta di I/O	Accumulatore	Registro B
		XXXXXXXX		?
IN	A,(INVD)	XXXXXXXX →	XXXXXXXX	?
CPL		XXXXXXXX	X̄X̄X̄X̄X̄X̄X̄X̄	?
			∧ 00010000	
AND	10H	XXXXXXXX	000X̄0000	?
LD	B,A	XXXXXXXX	000X̄0000 →	000X̄0000
IN	A,(INVD)	XXXXXXXX →	XXXXXXXX	000X̄0000
			∧ 11101111	
AND	0EFH	XXXXXXXX	XXX0XXXX	000X̄0000
			∨ 000X̄0000 ←	
OR	B	XXXXXXXX	XXXX̄XXXX	000X̄0000
OUT	(INVD),A	XXXX̄XXXX ←	XXXX̄XXXX	000X̄0000

INVERSIONE DEL BIT USANDO XOR

La procedura data sopra dimostra una tecnica preziosa — denominata mascheratura del bit. Tuttavia per la funzione specificata essa è troppo complicata. **Ecco una sequenza d'istruzioni più semplice che realizza la stessa inversione del bit:**

```
IN   A,(INVD)    ; Ingresso nell'Accumulatore dalla porta di I/O INVD
XOR  10H         ; Complemento del bit 4, salvando tutti gli altri bit
OUT  (INVD),A    ; Uscita dell'Accumulatore nella porta di I/O INVD
```

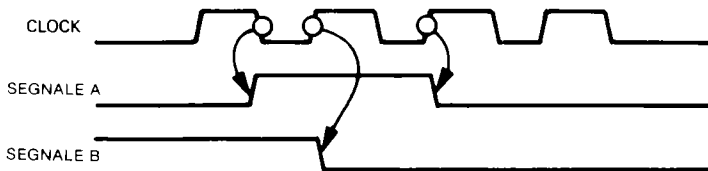
In questa sequenza di istruzioni noi usiamo la funzione OR esclusivo e la maschera appropriata per invertire il bit desiderato e preservare gli altri. La tavola della verità dell'OR esclusivo mostra che XOR con 1 inverte il bit, mentre XOR con 0 salva il valore del bit:

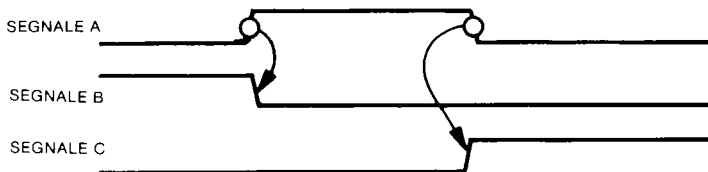
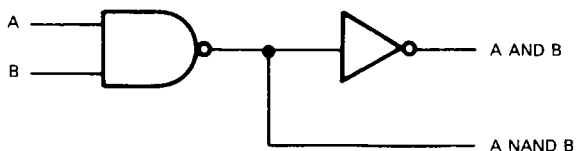
Y	X	X∨̶Y	
0	0	0	$X \veebar 0 = X$
0	1	1	
1	0	1	$X \veebar 1 = \overline{X}$
1	1	0	

Nella programmazione come nel progetto logico con componenti discreti, ci sarà spesso più di un modo per implementare la stessa funzione.

TEMPORIZZAZIONE DEGLI EVENTI

LOGICA SINCRONA

In una implementazione di logica digitale, gli eventi possono essere sincroni col tempo, riferiti ad un segnale di clock:

LOGICA ASINCRONA

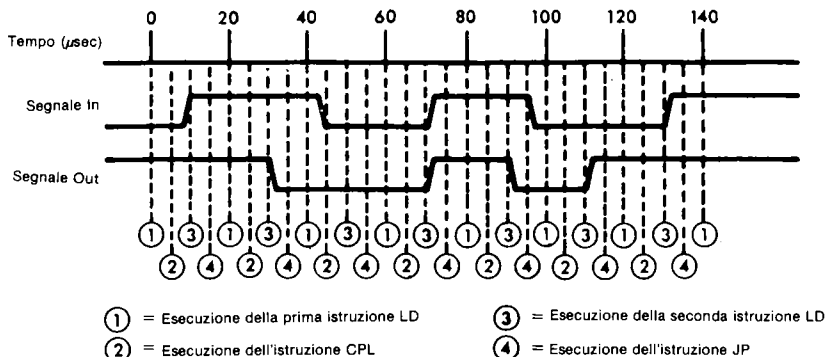
o asincroni, riferiti al segnale di uscita di un dispositivo che cambia stato, dando così l'avvio al cambiamento di stato di un altro dispositivo:

Le porte semplici tuttavia, sono dispositivi continui. Consideriamo la seguente semplice sequenza logica:

TEMPO DI ASSESTAMENTO DI UNA PORTA

L'invertitore di segnale inverte continuamente il suo ingresso; l'unico ritardo tra i cambiamenti di stato dei segnali d'ingresso e d'uscita è un tempo di assestamento di circa 10 nanosecondi.

In un sistema a microcalcolatore, tuttavia, si devono eseguire tre istruzioni prima che un segnale d'uscita possa rispecchiare un cambiamento di stato di un segnale di ingresso.

Nell'improbabile caso che il sistema a microcalcolatore stia emulando un invertitore e non stia facendo nient'altro, la sequenza delle istruzioni dell'invertitore potrebbe essere continuamente rieseguita come segue:

```
LOOP:  LD    A,(INVD)     ; Carica l'Accumulatore da INVD
       CPL                ; Complementa l'Accumulatore
       LD    (INVD),A     ; Memorizza il contenuto dell'Accumulatore in INVD
       JP    LOOP         ; Riesegue la sequenza dell'invertitore di segnale
```

Occorreranno circa 20 microsecondi per eseguire una sola volta il loop di istruzioni dell'invertitore di segnale, in funzione della frequenza del clock del microcalcolatore. Facendo in modo che il periodo tra i cambiamenti degli stati del segnale d'ingresso non sia mai minore di 20 microsecondi, l'invertitore di segnale implementato su microcalcolatore lavorerà sempre. Ma **ci può essere un ritardo anche di 30 microsecondi tra un cambiamento di stato del segnale d'ingresso e la conseguente risposta del segnale d'uscita.** Ciò può essere illustrato come segue:

Nella precedente illustrazione, le quattro istruzioni sono state mostrate dividendo 20 microsecondi egualmente, così che ogni istruzione è eseguita in 5 microsecondi. In realtà non è così. Il Capitolo 6 dà i tempi di esecuzione delle istruzioni; vedrete, per esempio, che l'istruzione CPL richiede un tempo di esecuzione considerevolmente minore delle altre tre istruzioni. Trascureremo per il momento questo dettaglio per concentrarci sul concetto che — **dobbiamo prestare molta attenzione alle sequenze degli eventi in un sistema a microcalcolatore.**

Lo stato del "Segnale In" al tempo ① (quando si esegue l'istruzione LD A,(INVD)) che, come bit, è portato nel sistema del microcalcolatore, non ha alcun riferimento a quando e a come il "Segnale In" cambia stato.

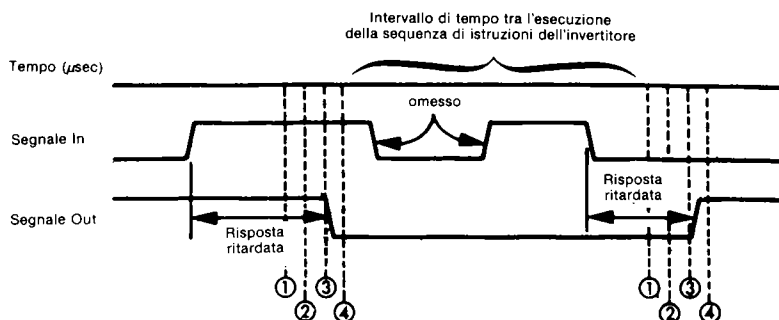
La reale inversione del bit avviene al tempo ②

Il bit invertito è convertito nel "Segnale Out" al tempo ③ quando si esegue la istruzione LD (INVD),A.

In tale modo, la temporizzazione del "Segnale Out" può differire considerevolmente dalla temporizzazione del "Segnale In".

Quando la sequenza di istruzioni dell'invertitore di segnale è solo una piccola parte di un più grande programma su un microcalcolatore, sorgono problemi più seri. In queste circostanze, possono trascorrere molti millisecondi tra esecuzioni ripetute della sequenza di istruzioni dell'invertitore. Se ciò fosse lasciato al caso, si potrebbero perdere completamente le inversioni del segnale. Nel caso migliore ci potrebbero essere considerevoli ritardi tra il cambiamento di stato del segnale d'ingresso e la conseguente risposta del segnale d'uscita. Questa situazione può essere illustrata come in figura a pagina successiva.

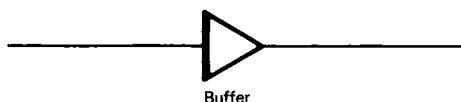
Nuovamente, ① , ② , ③ , e ④ identificano rispettivamente l'esecuzione delle istruzioni LD, CPL, LD e JP.

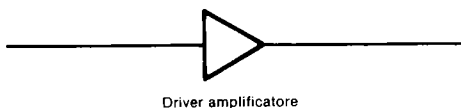
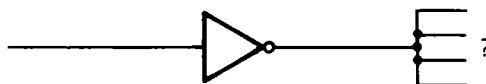
Dopo aver messo in rilievo l'importanza della temporizzazione in un sistema a microcalcolatore e le conseguenze di una temporizzazione insufficiente, abbandoneremo per il momento l'argomento. Ciò perchè **i problemi di temporizzazione si riducono notevolmente quando si simulano intere sequenze logiche in contrapposizione a**

dispositivi singoli. Perciò, le soluzioni ai problemi di temporizzazione dovrebbero essere viste nel contesto di una intera simulazione di logica; noi non siamo ancora entrati a fondo in ciò.

BUFFER, AMPLIFICATORI E CARICHI DI SEGNALE

Dopo aver visto la temporizzazione, torniamo ora a qualche altro concetto fondamentale di logica digitale.

BUFFER **Un buffer di segnale aumenta il livello della corrente di segnale:**

Buffer

AMPLIFICATORE **Un driver amplificatore aumenta il livello della tensione di segnale:**

Driver amplificatore

FAN OUT **Ogni dispositivo ha un fan out ben definito.** Il fan out definisce il numero di carichi paralleli che possono essere collegati al segnale di uscita:

FAN IN I dispositivi logici avranno inoltre un fan in specificato, che indica il numero di carichi paralleli che possono essere collegati all'ingresso di un dispositivo.

Che cosa succede a questi concetti una volta che la vostra logica scompare in un programma di un microcalcolatore? La risposta è semplice: questi concetti scompaiono — insieme con la logica digitale.

FAN IN
FAN OUT
CARICHI TTL
BUFFERIZZAZIONE DEI SEGNALI

Ora ai pin effettivi di un dispositivo fisico a microcalcolatore il fan in e il fan out rimangono concetti legittimi; i segnali viaggiando tra i pin dei singoli dispositivi del microcalcolatore possono aver bisogno di essere bufferizzati e amplificati. Per esempio, il fan out di un dispositivo Z80 può essere come uno o due carichi Transistor-Transistor Logic (TTL); ciò significa che se si collegano ad un segnale di uscita più di uno o due dispositivi simili, il segnale di uscita avrà una potenza insufficiente per trasmettere segnali utilizzabili da tutti i dispositivi collegati. Quindi per tutte le più semplici configurazioni a microcalcolatore le linee di bus dovranno essere bufferizzate.

CORRENTE DI LEAKAGE

Quando si determina se le linee di bus devono essere bufferizzate, non si ignori la corrente di leakage. Per esempio se avete sedici dispositivi ROM connessi sul Bus del Sistema e se potete selezionare (e quindi connettere) un solo dispositivo per volta, non dovete supporre che il carico totale di segnale sia dovuto alla ROM selezionata. I quindici dispositivi ROM non selezionati presenteranno in uscita una corrente di leakage; solo ciò può richiedere una bufferizzazione del Bus di Sistema.

In un programma su un microcalcolatore, tuttavia, quando la logica è totalmente rappresentata da una sequenza di istruzioni di microcalcolatore, ci si occupa esclusivamente di bit — mai di livelli di tensione o di corrente. Il fan in è infinito, dal momento che lo stato di un bit può essere il risultato di un numero qualsiasi di elaborazioni logiche. Il fan out è infinito perchè si può leggere lo stato di un bit quando si vuole. I buffer e gli amplificatori perdono di significato, poichè un bit non ha qualità equivalenti alla tensione o alla corrente. Un bit offre una soluzione pura e definita.

Diamo un'altra occhiata all'invertitore di segnale, quando è simulato dal microcalcolatore.

Faremo un passo concettuale gigante e supporremo che l'invertitore di segnale sia nascosto in una sequenza logica, così che non viene generato nessun segnale di ingresso o di uscita su alcun pin del dispositivo di microcalcolatore. In altre parole, l'invertitore di segnale diventa una piccola parte di una più vasta funzione di trasferimento.

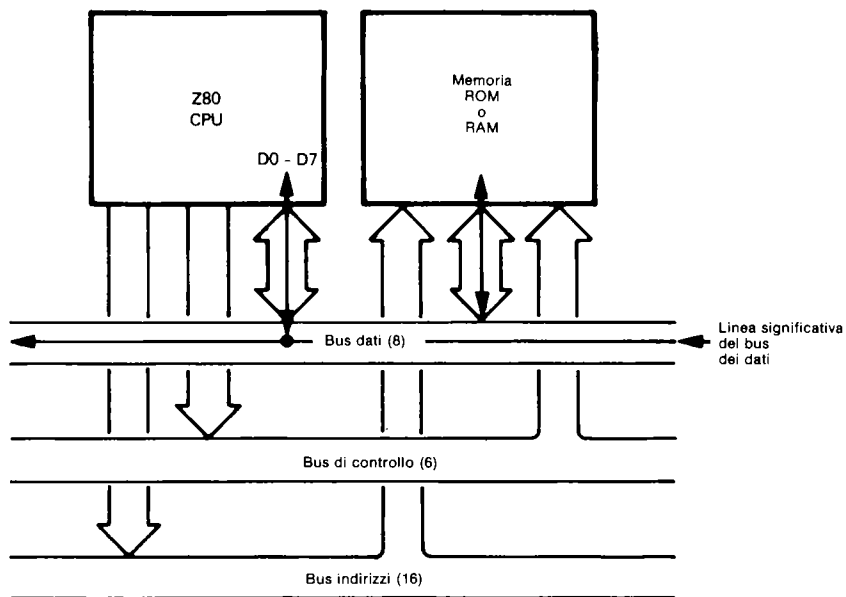
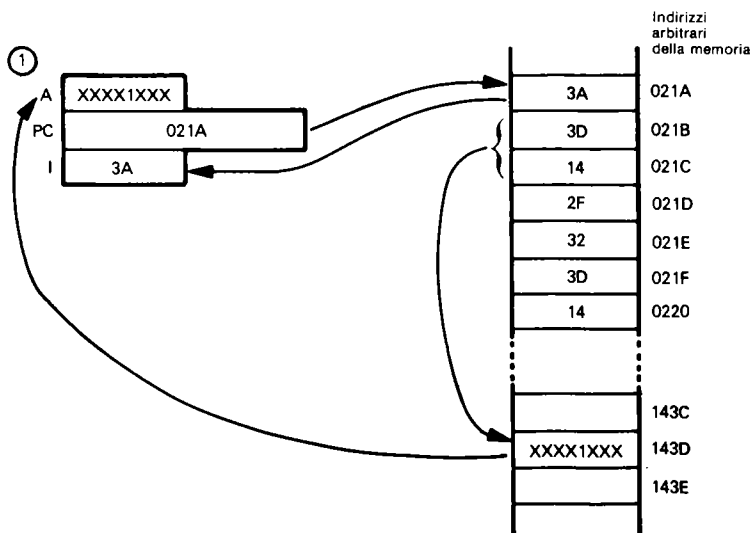
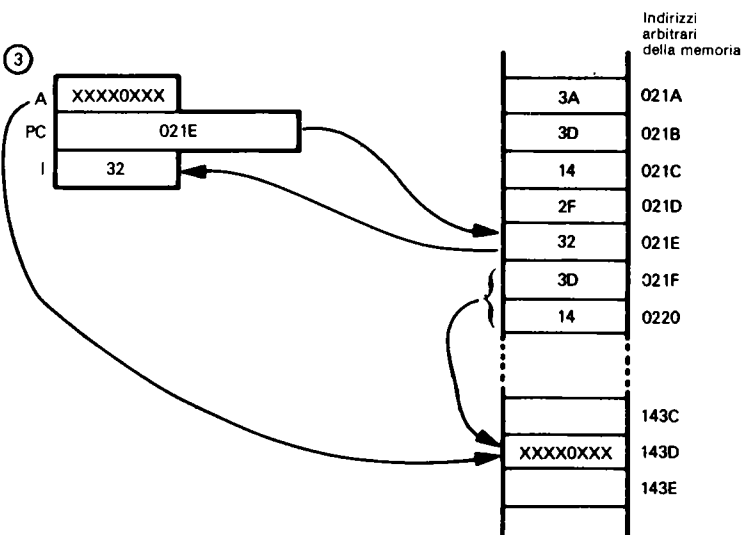
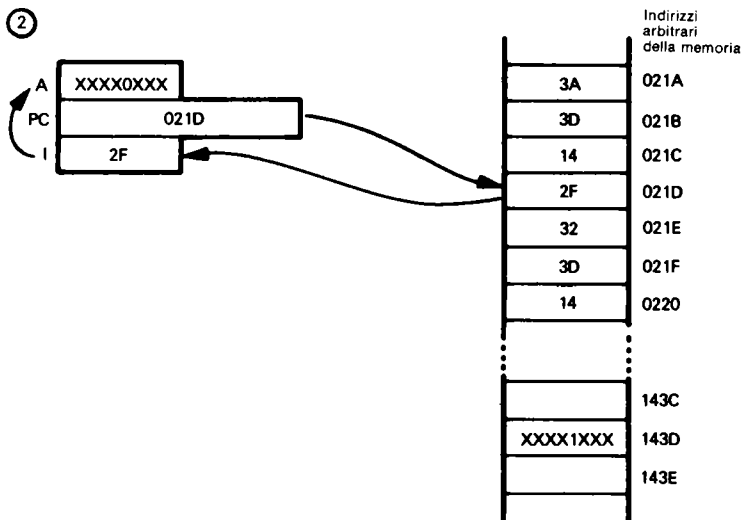
L'ingresso dell'invertitore di segnale è un bit creato da una logica precedente.

L'uscita dall'invertitore di segnale è un altro bit che diventa l'ingresso alla logica seguente.

COMPLEMENTAZIONE DI UN BYTE DI MEMORIA

La logica esterna al sistema a microcalcolatore non fornisce l'ingresso dell'invertitore come un segnale che arriva a un pin del dispositivo del microcalcolatore, nè permette che il segnale invertito sia trasmesso a una logica esterna attraverso un pin del dispositivo del microcalcolatore. Piuttosto l'interfaccia tra la logica esterna e il sistema a microcalcolatore è situata in qualche punto significativamente prima dell'invertitore di segnale. **Il nostro invertitore di segnale ora può essere rappresentato da queste tre stesse istruzioni:**

```
LD   A,(INVD)     ; Carica l'Accumulatore da INVD
CPL               ; Complementa
LD   (INVD),A     ; Memorizza il contenuto dell'Accumulatore in INVD
```

La sorgente e la destinazione diventano bit di dati di memoria; ciò può essere illustrato come segue:

In termini di contenuti di memoria e di registri di CPU, la sequenza dell'invertitore di segnale procede come segue:

Con riferimento all'illustrazione precedente, la lettera A identifica l'Accumulatore primario della CPU Z80. PC rappresenta il Contatore di Programma, e I rappresenta il registro delle istruzioni.

Il contenuto del byte $143D_{16}$ della memoria dati e dell'Accumulatore sono rappresentati in formato binario. X rappresenta un bit qualsiasi. Noterete che si è scelto arbitrariamente il bit 3 come bit significativo.

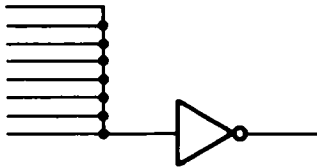
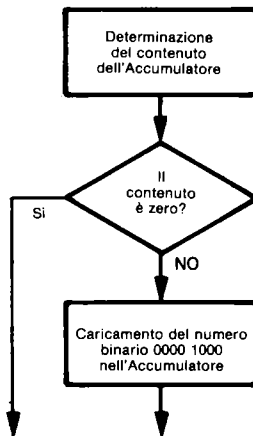
Nel passo ①, si esegue l'istruzione LD A,(INVD). Questa istruzione fa sì che il contenuto del byte $143D_{16}$ della memoria dati sia caricato nell'Accumulatore.

Durante il passo ② si esegue l'istruzione CPL. Ciò fa sì che il contenuto dell'Accumulatore sia complementato.

Durante il passo ③ , il contenuto dell'Accumulatore è ricaricato nel byte di memoria di $143D_{16}$.

L'inversione del segnale è stata simulata invertendo il contenuto del bit 3 (insieme con gli altri bit) del byte $143D_{16}$ della memoria dati.

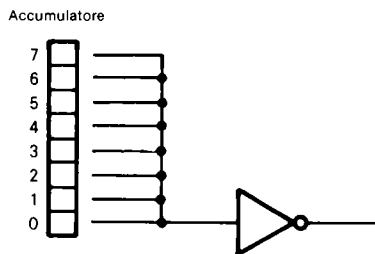
FAN IN IN PROGRAMMI DI MICROCALCOLATORI

Da dove proviene l'ingresso dell'invertitore? La risposta è: da un bit del dato di memoria. **Supponiamo, per illustrare ciò, che l'ingresso dell'invertitore sia l'OR di otto segnali.** Non potremmo mettere in OR cablato questi otto segnali per creare un invertitore come segue:

Ma, **supponendo che gli otto segnali siano rappresentati dal contenuto degli otto bit dell'Accumulatore, non avremo problemi generando l'ingresso dell'invertitore mediante la seguente sequenza logica:**

Il fan in nella logica è implementato dalla seguente sequenza d'istruzioni:

```
; Supponiamo che gli otto segnali siano nell'Accumulatore
; Ognuno sia rappresentato da un solo bit dell'Accumulatore
        AND     A           ; AND dell'Accumulatore con sé stesso per posiziona-
                            ; re il flag di stato
```

```
        JR      Z,NEXT     ; L'Accumulatore contiene zero. Il segnale d'ingresso
                           ; deve essere zero.
        LD      A,8        ; L'Accumulatore contiene un dato non nullo. Il se-
                           ; gnale d'ingresso deve essere 1
NEXT    LD      (INVD),A   ; Salva l'ingresso dell'invertitore
```

La sequenza d'istruzioni precedente è una implementazione diretta con programma su microcalcolatore di un OR cablato di otto segnali. Esaminiamo come funziona la logica delle istruzioni.

Supponiamo che gli otto segnali d'ingresso siano inizialmente rappresentati dallo stato degli otto bit dell'Accumulatore:

Supponiamo inoltre che, in accordo con l'illustrazione precedente, il bit 3 del byte del dato sarà il bit significativo del segnale dell'invertitore.

Poichè l'ingresso dell'invertitore è l'OR cablato di otto segnali, la logica del programma deve posizionare ad 1 il bit 3 dell'Accumulatore se nessun bit dell'Accumulatore è zero; il bit 3 dell'Accumulatore deve essere posto a zero se tutti i bit dell'Accumulatore sono a zero. Il contenuto dell'Accumulatore è allora memorizzato nel byte della memoria dei dati rappresentato dalla label INVD. Con riferimento alla illustrazione precedente, INVD è una label rappresentata dal byte di memoria $143D_{16}$.

Ciò spiega come funziona la sequenza di quattro istruzioni illustrata sopra:

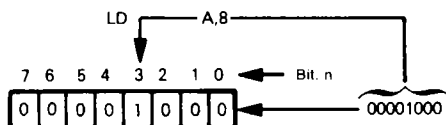
DETERMINAZIONE DELLO STATO FACENDO L'AND DI UN REGISTRO CON SE STESSO

Non conosciamo che cosa contiene inizialmente l'Accumulatore, così dobbiamo determinare il suo contenuto posizionando opportunamente i flag di stato della CPU. Per fare ciò, facciamo l'AND del contenuto dell'Accumulatore con sè stesso. Il fare l'AND dell'Accumulatore con sè stesso non cambia il contenuto dell'Accumulatore, ma vengono posizionati i flag di stato. A noi interessa solo lo stato Zero, che sarà posto a 1 se l'AND dell'Accumulatore con sè stesso genera un risultato nullo; il flag di stato Zero sarà posto a zero negli altri casi.

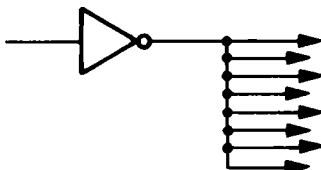
Ma l'AND dell'Accumulatore con sè stesso sarà zero solo se l'Accumulatore contiene zero:

X	Y	X Λ Y	
0	0	0	→ 0 Λ 0 = 0
0	1	0	} non applicabile
1	0	0	
1	1	1	→ 1 Λ 1 = 1

In tale modo, dopo l'esecuzione dell'istruzione AND, se lo stato Zero è 1 allora il bit 3 dell'Accumulatore deve essere già zero, come vogliamo che esso sia. Non è richiesta alcuna operazione e si salta all'istruzione LED (INVD),A.

Se il bit dello stato Zero era zero, allora uno o più bit dell'Accumulatore sono diversi da zero. L'istruzione LD A,8 carica 1 nel bit 3 dell'Accumulatore:

Infine, si esegue l'istruzione LD (INVD),A per caricare il segnale di ingresso dell'invertitore nell'appropriato byte della memoria dati.

Supponiamo ora che l'uscita dell'invertitore sia distribuita a numerosi successivi dispositivi.

La logica seguente rappresenta il fan out che non è possibile:

FAN OUT IN PROGRAMMI SU MICROCALCOLATORE

In un programma di microcalcolatore, l'intero concetto di fan out scompare. Si può accedere all'uscita un numero indefinito di volte con la semplice riesecuzione di una istruzione LD:

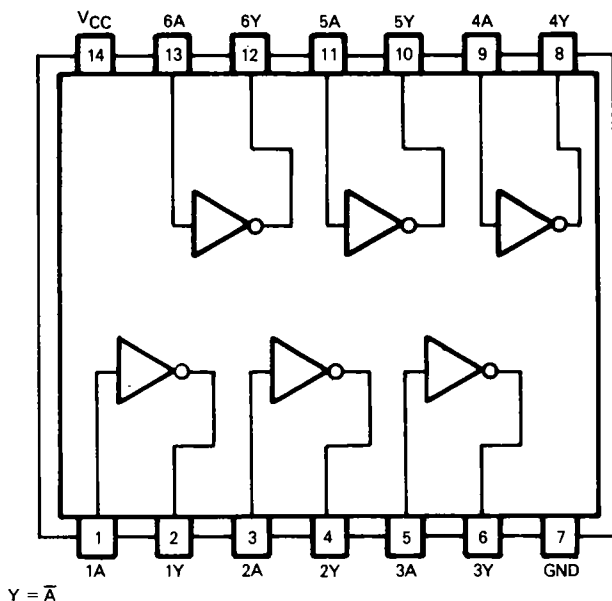
```
LD    A,(INVD)    ; Carica l'uscita dell'invertitore nell'Accumulatore
—
—
—
LD    A,(INVD)    ; Carica l'uscita dell'invertitore nell'Accumulatore
—
—
—
LD    A,(INVD)    ; Carica l'uscita dell'invertitore nell'Accumulatore
—
—
—
LD    A,(INVD)    ; Carica l'uscita dell'invertitore nell'Accumulatore
—
—
—
LD    A,(INVD)    ; Carica l'uscita dell'invertitore nell'Accumulatore
```

Che cosa si può dire sugli amplificatori e sui buffer? Chiaramente, nel contesto dei dati binari immagazzinati in memoria, essi non hanno alcuni significato. Se gli amplificatori ed i buffer sono presenti a causa delle caratteristiche elettriche dei chip di memoria e del processore, ciò non ha niente a che fare con la funzione logica che deve essere implementata da un programma su microcalcolatore.

SIMULAZIONE CON MICROCALCOLATORE DEGLI INVERTITORI SESTUPLI 7404/7405/7406

Questi tre invertitori sestupli differiscono solo per le loro caratteristiche elettriche:

- **il 7404 è un semplice invertitore sestuplo**
- **il 7405 è un invertitore sestuplo con uscite a collettore aperto**
- **il 7406 è un invertitore sestuplo buffer/driver con uscite a collettore aperto ed alta tensione.**

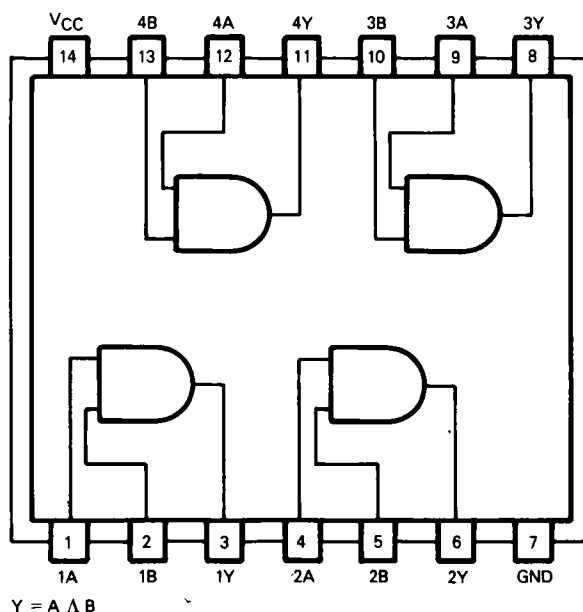
Poichè questi tre dispositivi differiscono solo nelle loro caratteristiche elettriche, essi sono identici in una simulazione con linguaggio assembly di un microcalcolatore. Guardiamo il 7404. Esso è costituito da sei invertitori di segnale, che possono essere illustrati come segue:

La sequenza d'istruzioni per rappesentare un invertitore sestuplo è identica alla sequenza d'istruzioni per un invertitore singolo di segnale, formata da tre istruzioni, poichè i microcalcolatori Z80 sono dispositivi di otto bit paralleli. Che vi piaccia o no, questa sequenza d'istruzioni dell'invertitore inverte otto bit indipendenti. Perciò gli invertitori sestupli possono essere rappresentati con una sequenza d'istruzioni in un microcalcolatore come segue:

```
LD    A,(INVD)   ; Carica l'Accumulatore da INVD
CPL              ; Complementa
LD    (INVD),A   ; Memorizza i contenuti dell'Accumulatore in INVD
```

Identificheremo i bit significativi arbitrariamente, come implicito nell'invertitore sestuplo come segue:

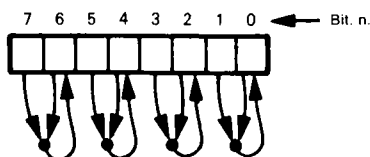
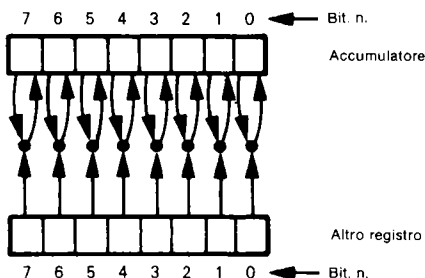
Notate che la scelta precedente dei bit significativi è completamente arbitraria. Non c'è assolutamente nessun argomento filosofico o pratico per favorire un'assegnazione dei bit piuttosto che un'altra.

SIMULAZIONE CON MICROCALCOLATORE DELLE QUADRUPLE PORTE 7408/7409 DI TIPO AND POSITIVO A DUE INGRESSI

Questi due dispositivi forniscono quattro porte AND indipendenti a due ingressi ed una sola uscita, e possono essere illustrati come segue:

Il 7409 ha le uscite a collettore aperto, che lo differenzia dal 7408. Questa differenza non ha significato in una simulazione in un programma su microcalcolatore; perciò si possono vedere i due dispositivi come identici.

FUNZIONI A DUE INGRESSI

Dal punto di vista del programmatore di microcalcolatore la differenza più significativa tra la porta AND 7408 e l'invertitore 7404 non è la funzione logica; piuttosto il fatto che il 7408 è un dispositivo a due ingressi. Concettualmente possiamo immaginare il 7404 simulato in uno dei due seguenti modi:

1) Gli otto segnali d'ingresso sono caricati nel registro Accumulatore della CPU. Ogni bit di posto pari è messo in AND col bit alla sua destra. Il risultato è depositato nei bit di posto pari per ogni coppia di bit:

2) I due insiemi di quattro ingressi sono caricati nell'Accumulatore e in un altro registro. Il risultato è rimesso nell'Accumulatore:

Quando esaminerete l'insieme delle istruzioni del microcalcolatore Z80, troverete che il secondo metodo per simulare un 7408 è quello naturale. La sequenza d'istruzioni richiesta è questa:

```
LD    A,(SRCA)    ; Carica il primo insieme di ingressi da SRCA
LD    B,A         ; Salvataggio nel Registro B
LD    A,(SRCB)    ; Carica il secondo insieme di ingressi da SRCB
AND   B           ; AND di A con B
LD    (DST),A     ; Salvataggio del risultato in DST
```

ASSEGNAZIONE DELLE LABEL AL PROGRAMMA SORGENTE

Se l'uso delle label SRCA, SRCB e DST vi confonde ancora, prendiamoci un minuto per chiarirle. Eventualmente avrete una certa quantità di memoria, che può variare da 256 byte a 65536 byte. Ognuna delle label SRCA, SRCB e DST identifica un solo byte di memoria. Quando state scrivendo un programma sorgente, il byte esatto di memoria identificato da ogni label non è importante. Eventualmente quando voi assemblate il vostro programma sorgente, la compilazione dell'assembler vi stamperà una mappa della memoria. La mappa della memoria identificherà il byte esatto di memoria associato ad ogni label che voi avete usato. Esaminando la mappa di memoria, sarete capaci di determinare se tutte le assegnazioni di label sono o non sono valide. Se qualche assegnazione di label non è valida, dovrete fare l'azione appropriata. L'azione appropriata può comportare un'aggiunta di memoria alla vostra configurazione di microcalcolatore, o dovrete riscrivere il vostro programma sorgente così che si renda più efficiente l'uso della memoria a vostra disposizione. Il problema delle assegnazioni delle label e della memoria è irrilevante a questo punto della discussione. Immaginiamo semplicemente ogni label come indirizzo di un solo byte di memoria. Non preoccupatevi di quale byte di memoria sarà alla fine indirizzato e il vostro problema scomparirà.

La sequenza d'istruzioni per simulare il 7408 illustrata sopra non rappresenta in nessun modo l'unica maniera in cui si può simulare un 7408.

Consideriamo dapprima alcune minime variazioni. Si potrebbero usare i Registri C, D, E, H o L della CPU invece del Registro B per conservare il secondo dato d'ingresso. Eccone un esempio:

```
LD    A,(SRCA)    ; Carica il primo insieme d'ingressi da SRCA
LD    C,A         ; Salvataggio nel Registro C
LD    A,(SRCB)    ; Carica il secondo insieme di ingressi da SRCB
AND   C           ; AND di C con A
LD    (DST),A     ; Salvataggio del risultato in DST
```

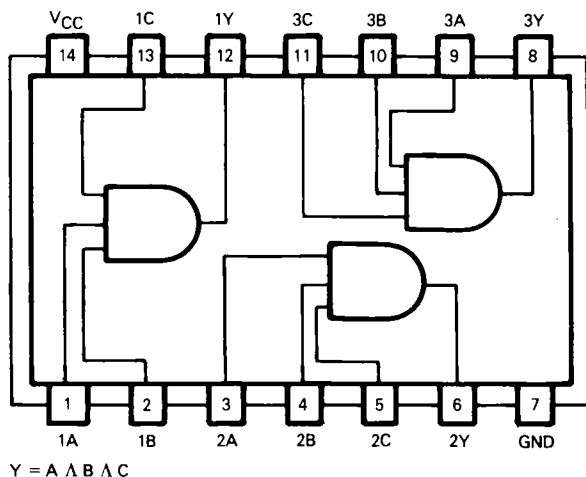
INDIRIZZAMENTO IMPLICITO DI MEMORIA

Non è da incoraggiare l'uso dei Registri H o L per conservare il secondo ingresso. L'uso principale di questi due registri è quello di conservare un indirizzo di un dato di memoria. Per esempio le istruzioni LD A,(SRCA); LD A,(SRCB); e LD (DST),A potrebbero essere sostituite come segue:

```
LD    HL,SRCA     ; Carica l'indirizzo del primo insieme di ingressi in H,L
LD    A,(HL)      ; Carica il primo insieme di ingressi in A
LD    HL,SRCB     ; Carica l'indirizzo del secondo insieme di ingressi in
                  ; H, L
AND   (H,L)       ; AND fra il secondo insieme di ingressi con A
LD    HL,DST      ; Carica l'indirizzo di destinazione in H, L
LD    (HL),A      : Memorizza il risultato in DST
```

LA SIMULAZIONE CON MICROCALCOLATORE DI UN 7411 TRIPLA PORTA AND POSITIVO A TRE INGRESSI

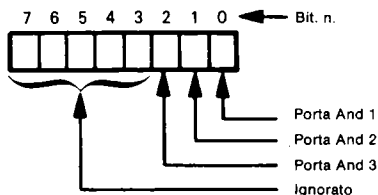
La differenza principale tra una porta AND 7411 e una porta AND 7408 è il numero di segnali d'ingresso. Il 7411 genera tre segnali d'uscita, ognuno dei quali è l'AND di tre ingressi:

$Y = A \wedge B \wedge C$

FUNZIONI A TRE INGRESSI

Di nuovo siamo di fronte a delle scelte. Possiamo caricare i tre insiemi d'ingressi in tre registri della CPU (l'Accumulatore e altri due registri), quindi effettuare due AND prima di mettere a posto il risultato:

```
ONE     LD   A,(SRCA)   ; Carica il primo insieme di ingressi da SRCA
TWO     LD   B,A        ; Salvataggio nel Registro B
THREE   LD   A,(SRCB)   ; Carica il secondo insieme di ingressi da SRCB
FOUR    LD   C,A        ; Salvataggio nel Registro C
FIVE    LD   A,(SRCC)   ; Carica il terzo insieme di ingressi da SRCC
SIX     AND  B          ; AND di B con A
SEVEN   AND  C          ; AND di C con A
EIGHT   LD   (DST),A    ; Salvataggio del risultato in DST
```

Alle istruzioni nella sequenza precedente sono state assegnate label ben definite per rendere più facile da capire la descrizione che segue. Le istruzioni non hanno bisogno di label per soddisfare le necessità di un programma sorgente in linguaggio assembly.

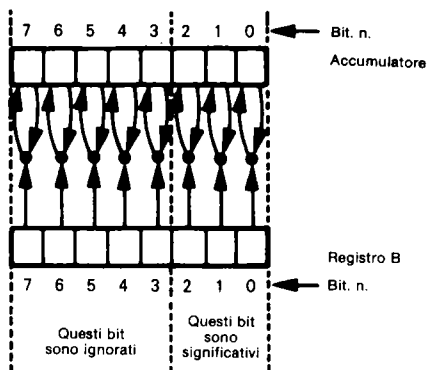
Quando si esegue l'istruzione ONE, si carica un valore di 8 bit nell'Accumulatore dal byte di memoria indirizzato dalla label SRCA. Supporremo che gli ingressi della porta AND siano rappresentati come segue:

Capirete che l'assegnazione dei bit dei dati sopra illustrata è completamente arbitraria. E' necessario solamente che tutti gli ingressi successivi siano coerenti.

Dopo l'esecuzione dell'istruzione ONE, il primo insieme di ingressi è nell'Accumulatore. L'Accumulatore è il solo registro della CPU nel quale i dati possono essere caricati se si usa un indirizzamento diretto. Il primo insieme di ingressi deve quindi essere salvato in un altro registro, cosí che l'Accumulatore sia libero per essere caricato con un secondo insieme di ingressi. L'istruzione TWO sposta il contenuto dello Accumulatore nel registro B.

Le istruzioni THREE e FOUR caricano il secondo insieme di ingressi nell'Accumulatore, poi lo spostano nel registro C. Supponiamo che l'assegnazione dei bit del secondo insieme di ingressi sia identico all'assegnazione illustrata sopra per il primo ingresso.

Il terzo ed ultimo insieme di ingressi è caricato nell'Accumulatore dall'istruzione FIVE.

L'istruzione AND fa l'AND del contenuto del registro di CPU con il contenuto dell'Accumulatore, lasciando il risultato nell'Accumulatore. L'istruzione SIX effettua il primo AND come illustrato in figura di pagina 2-27.

L'istruzione SEVEN effettua la seconda operazione di AND. In quest'istante avviene

l'AND tra l'Accumulatore e il Registro C. L'Accumulatore inizialmente mantiene il risultato dell'AND con B, illustrato sopra. Dopo l'esecuzione della istruzione SEVEN, l'AND dei tre ingressi è nell'Accumulatore.

L'istruzione EIGHT riporta il risultato finale nel byte di memoria indirizzato dalla label DST. La simulazione della porta AND 7411 è completata.

Ora consideriamo una simulazione alternativa delle porte AND 7411. Possiamo caricare il primo ingresso nell'Accumulatore e il secondo ingresso in un altro registro. Dopo aver fatto l'AND di questi due ingressi, possiamo caricare il terzo ingresso nello stesso "altro" registro, farne l'AND col risultato del primo AND, quindi riportare il risultato:

```
ONE     LD    A,(SRCA)    ; Carica il primo insieme di ingressi da SRCA
TWO     LD    B,A         ; Salvataggio nel Registro B
THREE   LD    A,(SRCB)    ; Carica il secondo insieme di ingressi da SRCB
FOUR    AND   B           ; AND tra B ed A, il risultato é in A
FIVE    LD    B,A         ; Salvataggio del risultato in B
SIX     LD    A,(SRCC)    ; Carica il terzo insieme di ingressi da SRCC
SEVEN   AND   B           ; AND di B con A
EIGHT   LD    (DST),A     ; Salvataggio del risultato in DST
```

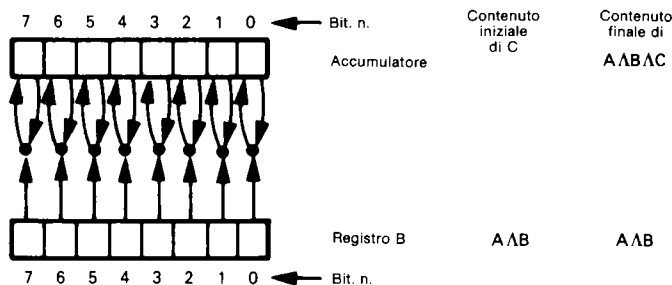
Confrontiamo questa seconda simulazione della porta AND 7411 con la prima simulazione. Le istruzioni ONE, TWO e THREE sono identiche a quelle della prima simulazione. Dopo che si sono eseguite queste tre istruzioni, un insieme di ingressi si trova nel Registro B ed un secondo insieme di ingressi nell'Accumulatore. Questa è la situazione:

Gli ingressi A sono nel Registro B

Gli ingressi B sono nell'Accumulatore

Ora, invece di portare immediatamente il terzo insieme di ingressi in un registro della CPU, eseguiamo l'istruzione FOUR, che genera l'AND dei primi due ingressi. Poichè questo AND è generato nell'Accumulatore, noi salviamo il risultato nel Registro B eseguendo l'istruzione FIVE. L'effetto puro è questo:

$A \wedge B$ nel Registro B

L'istruzione SIX ora carica il terzo insieme di ingressi nell'Accumulatore.
L'istruzione SEVEN fa l'AND del terzo insieme di ingressi con il risultato del primo AND, come segue:

L'istruzione EIGHT salva il risultato dall'Accumulatore nel byte di memoria indirizzato dalla label DST.

MINIMIZZAZIONE DEGLI ACCESSI A REGISTRI DELLA CPU

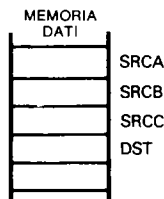
Qual'è la migliore simulazione delle porte AND 7411? Chiaramente la migliore è la seconda soluzione. C'è un problema non ovvio associato all'uso indiscriminato dei registri della CPU. Noi abbiamo deciso arbitrariamente che il Registro B conterrà un secondo ingresso. Dal momento che stiamo simulando le porte AND 7411 senza riguardo a ciò che precede o segue, la scelta del Registro B è arbitraria; la sua scelta non comporta nè vantaggi nè conseguenze.

CONFLITTI NELLA UTILIZZAZIONE DEI REGISTRI DELLA CPU

Invariabilmente una sequenza d'istruzioni come quella della simulazione delle porte AND 7411 è solo una piccola parte di una più grande. Dobbiamo ora preoccuparci se l'uso del Registro B per memorizzare il secondo ingresso interferirà con l'uso precedente o seguente del Registro B. Un errore di programmazione molto comune comporta conflitti di utilizzazione dei registri della CPU. Per esempio, che cosa succede se in alcuni passi logici precedenti si usa il Registro B per conservare un dato intermedio? Ora la simulazione del 7411 cancellerà il dato che era stato temporaneamente memorizzato in questo registro.

Per ridurre i conflitti dei registri della CPU, è sempre preferibile scegliere una sequenza di istruzioni che usa il minor numero possibile di registri, facendo in modo che non ci sia nessuna significativa penalizzazione. In questo caso non c'è alcuna penalizzazione significativa. Non si richiede un numero maggiore d'istruzioni per la simulazione delle porte AND 7411 usando solo il Registro B della CPU di quanto richiede l'uso dei Registri B e C. Perciò l'uso del solo Registro B è il metodo migliore.

INDIRIZZAMENTO IMPLICITO

Consideriamo ora la simulazione delle porte AND 7411 usando un indirizzamento implicito. Supponiamo che i tre ingressi delle porte AND siano memorizzati in byte sequenziali della memoria dati e che la destinazione segua l'ultimo byte della sorgente, come segue:

Ora usando un indirizzamento implicito, otteniamo la seguente sequenza di istruzioni:

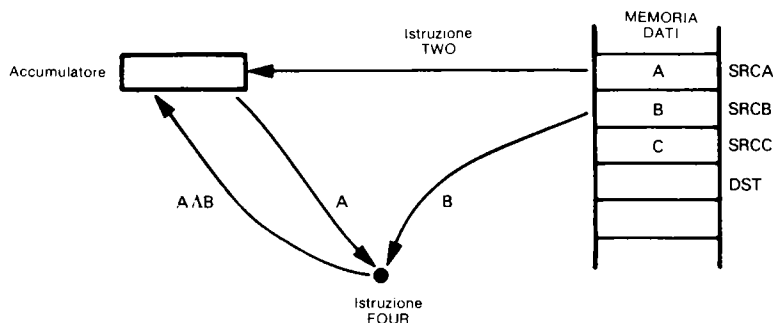
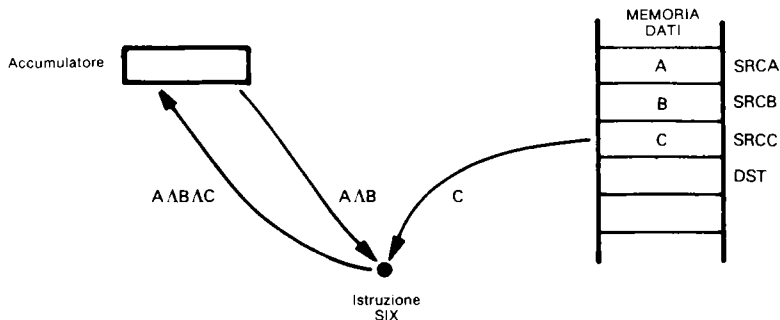
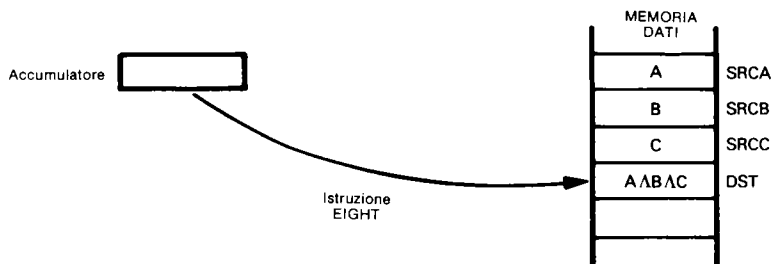
```
ONE     LD    HL,SRCA   ; Carica l'indirizzo della prima sorgente in HL
TWO     LD    A,(HL)    ; Carica la prima sorgente nell'Accumulatore
THREE   INC   HL        , Incrementa l'indirizzo implicito
FOUR    AND   (HL)      ; AND tra l'Accumulatore e la seconda sorgente
FIVE    INC   HL        ; Incrementa l'indirizzo implicito
SIX     AND   (HL)      ; AND tra l'Accumulatore e la terza sorgente
SEVEN   INC   HL        ; Incrementa l'indirizzo implicito
EIGHT   LD    (HL),A    ; Salvataggio del risultato
```

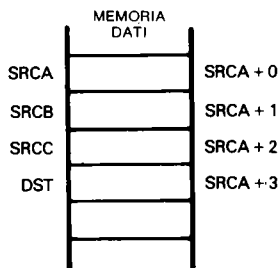
La sequenza delle istruzioni sarà eseguita così:

L'istruzione ONE carica l'indirizzo del primo byte della sorgente nei registri H ed L.

L'istruzione TWO sposta il contenuto del byte di memoria indirizzato da H ed L nell'Accumulatore.

L'istruzione THREE incrementa l'indirizzo a 16 bit dei registri H ed L, che ora indirizzano SRCB.

L'istruzione FOUR fa l'AND tra il contenuto dell'Accumulatore e la seconda sorgente, indirizzata dai registri H ed L. Il risultato è salvato nell'Accumulatore. Ciò può essere illustrato come segue:

Le istruzioni FIVE e SIX incrementano l'indirizzo implicito e ripetono l'operazione di AND, facendo questa volta l'AND del terzo ingresso con l'AND dei primi due ingressi. Ciò può essere illustrato come segue:

L'istruzione SEVEN incrementa ancora l'indirizzo in H ed L così che ora esso indirizza DST. L'istruzione EIGHT salva il risultato nella destinazione, come segue:

INDIRIZZAMENTO RELATIVO AD UNA BASE

Noi possiamo usare un indirizzamento relativo ad una base nella simulazione delle porte AND 7411. Come nell'esempio precedente, supponiamo che i tre ingressi siano memorizzati in byte sequenziali della memoria dati e che la destinazione segua l'ultimo byte della sorgente. Possiamo pensare ad ognuna di queste locazioni in termini della sua distanza relativa da SRCA:

Ecco la sequenza delle istruzioni:

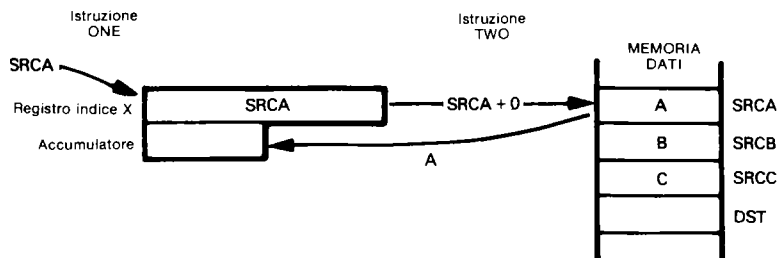
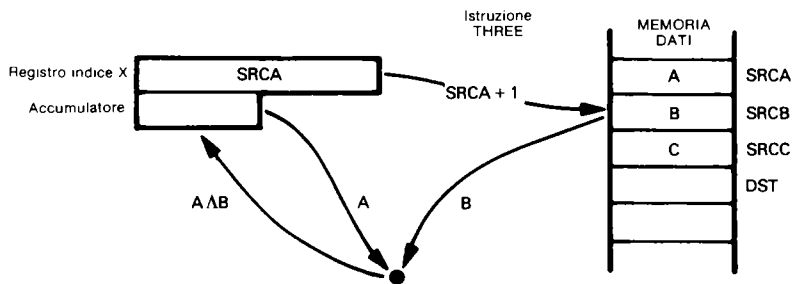
```
ONE     LD    IX,SRCA    ; Carica il primo indirizzo della sorgente in IX
TWO     LD    A,(IX+0)   ; Carica la prima sorgente nell'Accumulatore
THREE   AND   A,(IX+1)   ; AND tra l'Accumulatore e seconda sorgente
FOUR    AND   A,(IX+2)   ; AND tra l'Accumulatore e terza sorgente
FIVE    LD    (IX+3),A   ; Salvataggio del risultato
```

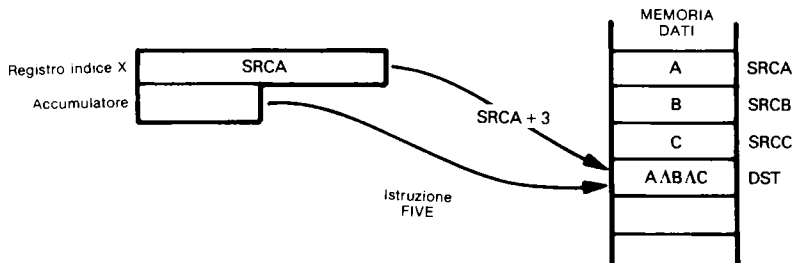
Finchè si ha a che fare con l'Accumulatore e la memoria dati, questa sequenza opera esattamente come la precedente. Tuttavia il registro degli Indirizzi è usato in un modo differente: invece di incrementare il registro prima del prossimo accesso di memoria, si somma un indice all'indirizzo di base, lasciando immutato il contenuto del registro. Questo è l'indirizzamento relativo ad una base descritto in An Introduction to Microcomputers: Volume I - Basic Concepts.

Ecco l'esecuzione della sequenza passo per passo.

L'istruzione ONE carica l'indirizzo del primo byte della sorgente nel Registro X.

Quando si esegue l'istruzione TWO, si somma l'indice 0 al contenuto del Registro

Indice X per ottenere l'indirizzo del primo byte della sorgente. Quel byte è poi spostato nell'Accumulatore. Ciò può essere illustrato come segue:

L'istruzione THREE fa l'AND del contenuto dell'Accumulatore col secondo byte della sorgente, che è indirizzato sommando l'indice 1 al contenuto del Registro Indice X. Ciò può essere illustrato come segue:

L'istruzione FOUR fa l'AND del contenuto dell'Accumulatore con il terzo byte della sorgente, che è indirizzato sommando l'indice 2 al contenuto del Registro Indice X.

L'istruzione FIVE carica l'Accumulatore nella locazione di memoria indirizzata sommando 3 al contenuto del Registro Indice X. In tale modo, l'AND tra i tre byte della sorgente è salvato nel byte di destinazione. Ciò può essere illustrato come segue:

Sebbene questa sequenza d'istruzioni abbia un numero minore di linee di codice della precedente, in realtà è meno efficiente, come dimostreremo di seguito. Questo non è un uso appropriato dell'indirizzamento relativo ad una base.

CONFRONTO DELLA UTILIZZAZIONE DELLA MEMORIA E DELLA VELOCITA' DI ESECUZIONE

Ora noi possediamo questi quattro programmi, ognuno dei quali simula porte AND 7411:

Il programma 1 usa un indirizzamento diretto e tre registri di CPU.
Il programma 2 usa un indirizzamento diretto e due registri di CPU.
Il programma 3 usa un indirizzamento implicito.
Il programma 4 usa un indirizzamento relativo ad una base.

Confrontiamo il numero di byte di programma oggetto richiesto per memorizzare ogni programma col numero di cicli di clock di CPU richiesto per eseguire ogni programma. I risultati sono elencati nella tabella 2-1. La tabella 2-1 include la mnemonica delle istruzioni per ogni programma, per aiutarvi nel seguire come tutti i byte del programmai oggetto e cicli di esecuzione siano stati calcolati. Vedere il Capitolo 6 per i dati di cui avete bisogno per verificare la tabella 2-1.

INDIRIZZAMENTO DIRETTO RISPETTO A INDIRIZZAMENTO IMPLICITO

I programmi 1 e 2 hanno una identica utilizzazione di memoria e velocità di esecuzione — cosa non sorprendente, poichè essi si differenziano per la sequenza secondo la quale le stesse istruzioni sono eseguite. **Il programma 3 adotta una filosofia completamente differente per la simulazione delle porte AND 7411, usando un indirizzamento di memoria implicito piuttosto che un indirizzamento diretto. Il risultato è drammatico. Si risparmiano sei byte di memoria e il programma viene eseguite nell'82% del tempo.** Ma il programma 3 pone una restrizione aggiuntiva alla simulazione; le sorgenti dei tre dati e la destinazione devono occupare quattro byte contigui della memoria dati. Il programma 4 ha un numero di linee di codice minore degli altri tre programmi, ma esso non risparmia byte ed ha il più lungo tempo di esecuzione. Inoltre esso restringe la locazione delle sorgenti e della destinazione dei dati. **L'indirizzamento relativo ad una base** è un accorgimento sofisticato che può salvare tempo e spazio di programma, ma **non è appropriato per questo particolare programma.**

Come classificheremo le tre opzioni di simulazione?

CLASSIFICAZIONE DELLE VARIAZIONI DEI PROGRAMMI

Lo schema sofisticato di indirizzamento del programma 4 non è adatto per questa applicazione. Abbiamo già concluso che il programma 2 batte il programma 1, perchè il programma 1 fa uso gratuito di un registro supplementare della CPU. Il programma 3 è chiaramente migliore del programma 2, ponendo la restrizione sulle locazioni dei dati della sorgente e della destinazione sia tollerabile.

Riguardo alla superiorità del programma 3 sul programma 2, vale la pena di notare ancora, come è stato verificato in An Introduction to Microcomputers: Volume I - Basic Concepts, che l'uso indiscriminato dell'indirizzamento diretto in applicazioni con microcalcolatore può essere costoso. L'indirizzamento implicito della memoria può sembrare primitivo per un programmatore con conoscenza di minicalcolatori o di grandi calcolatori, ma esso è economico.

Tabella 2-1. Confronto della utilizzazione di memoria e della velocità di esecuzione di un programma per la simulazione delle porte AND 7411

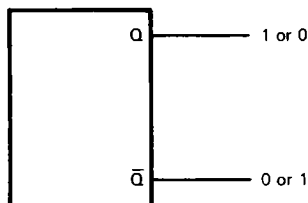
PROGRAMMA 1			PROGRAMMA 2			PROGRAMMA 3			PROGRAMMA 4		
MNEMONICA	BYTE	CICLI	MNEMONICA	BYTE	CICLI	MNEMONICA	BYTE	CICLI	MNEMONICA	BYTE	CICLI
LD[1]	3	13	LD[1]	3	13	LD[1]	3	10	LD[1]	4	14
LD[2]	1	4	LD[2]	1	4	LD[1]	1	7	LD[1]	3	19
LD[1]	3	13	LD[1]	3	13	INC	1	6	AND[1]	3	19
LD[2]	1	4	AND[2]	1	4	AND[1]	1	7	AND[1]	3	19
LD[1]	3	13	LD[2]	1	4	INC	1	6	LD[1]	3	19
AND[2]	1	4	LD[1]	3	13	AND[1]	1	7			
AND[2]	1	4	AND[2]	1	4	INC	1	6			
LD[1]	3	13	LD[1]	3	13	LD[1]	1	7			
TOTALE	16	68	TOTALE	16	68	TOTALE	10	56	TOTALE	16	90

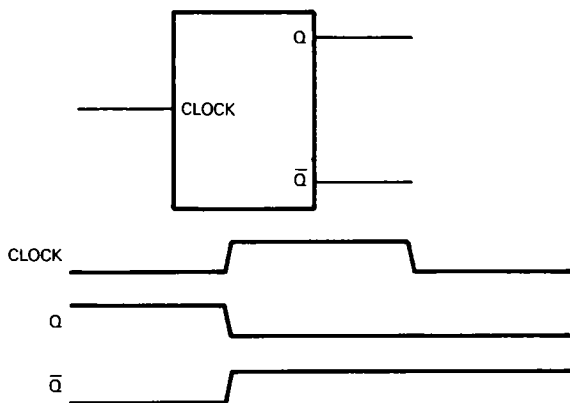
[1] Versione dell'istruzione Registro-Memoria
[2] Versione dell'istruzione Registro-Registro

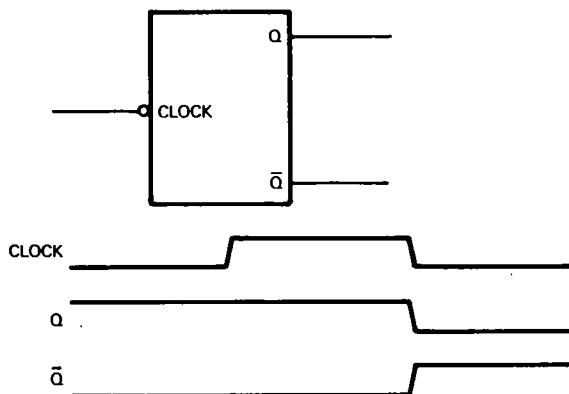
SIMULAZIONE CON MICROCALCOLATORE DI UN 7474, DOPPIO FLIP-FLOP DI TIPO D INNESCATO SUL FRONTE POSITIVO CON INGRESSI DI PRESET E DI AZZERAMENTO

Prima di guardare il flip-flop 7474 in particolare, consideriamo i flip-flop in generale. Dapprima diamone una definizione.

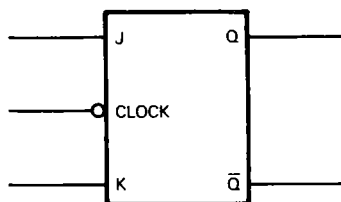
DESCRIZIONE LOGICO-DIGITALE DEI FLIP-FLOP

Un flip-flop è un dispositivo logico bistabile, cioè, un dispositivo che può essere in una di due condizioni stabili. I flip-flop 7474 hanno due uscite Q e $\overline{Q}$; in tale modo le due possibili condizioni stabili possono essere rappresentate come segue:

INNESCO SUL FRONTE POSITIVO

Un segnale di clock fa sì che il flip-flop cambi da una condizione stabile all'altra. Un flip-flop innescato sul fronte positivo cambia quando sente una transizione da zero ad uno del segnale di clock:

INNESCO SUL FRONTE NEGATIVO

Un flip-flop innescato sul fronte negativo cambia stato quando sente una transizione da uno a zero del segnale di clock. (Vedi pagina 2-35).

FLIP-FLOP JK

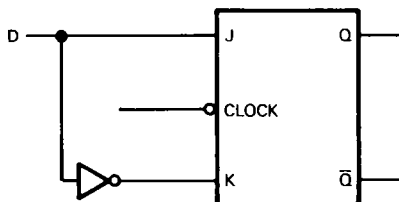
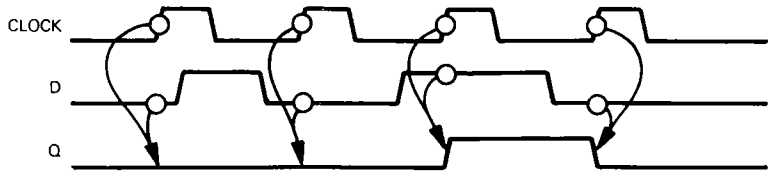
Un flip-flop JK pone un condizionamento alle uscite Q e $\overline{Q}$ che saranno generate dal fronte di trigger del prossimo clock come segue:

Stato di J e K nell'istante del segnale di clock		Uscite generate nell'istante del segnale di clock	
J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	Rimane nello stato precedente	
1	1	Cambia stato senza alcun riguardo allo stato precedente	

SEGNALE DI CLOCK

Nella tabella precedente, "il segnale di clock" sarà una transizione da zero ad uno per un dispositivo innescato sul fronte positivo; esso sarà una transizione da uno a zero per un dispositivo innescato sul fronte negativo. Questa definizione di "segnale di clock" si applica pure al flip-flop descritto successivamente.

FLIP-FLOP DI TIPO D

Invertendo un ingresso J per generare l'ingresso K, si ottiene un flip-flop di tipo D. Le caratteristiche di un flip-flop di tipo D sono:

Stato di J e K nell'istante del segnale di clock		Uscite generate nell'istante del segnale di clock	
J=D	K=$\overline{J}$	Q	$\overline{Q}$
1	0	1	0
0	1	0	1

Ecco un diagramma della temporizzazione di un flip-flop di tipo D innescato sul fronte positivo:

Un flip-flop di tipo D farà uscire perciò, le condizioni d'ingresso che esistevano allo impulso di clock precedente.

PRESET DEL FLIP-FLOP

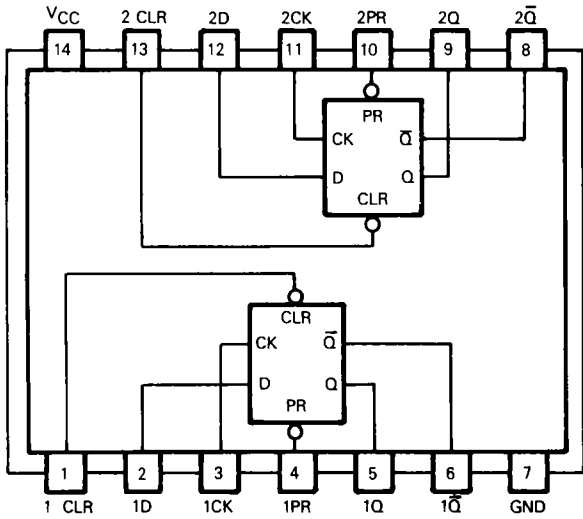
La presenza di un ingresso di Preset significa che il flip-flop può essere forzato in uscita con Q = 1 e $\overline{Q}$ = 0. La presenza del Preset forza questa condizione.

CLEAR DEL FLIP-FLOP

Un ingresso di azzeramento (Clear) è l'opposto di un ingresso di Preset. Quando è presente, il Clear forza Q = 0 e $\overline{Q}$ = 1.

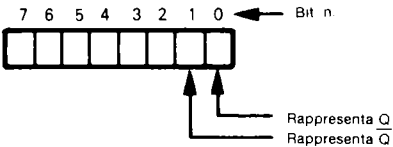
Combinando le definizioni date sopra, per un flip-flop di tipo 7474 si ha:

TABELLA DI FUNZIONAMENTO

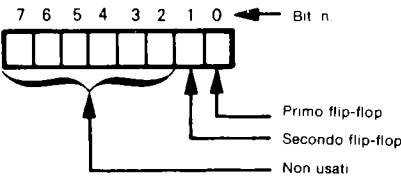
INGRESSI				USCITE	
1PR or 2PR	1CLR or 2CLR	1CK or 2CK	1D or 2D	1Q or 2Q	1$\overline{Q}$ or 2$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$

Nella tabella delle funzioni precedente, ↑ rappresenta una transizione del clock da zero ad uno. H* indica uno stato simile. Q_0 è lo stato precedente di Q. X significa "Don't care" (non considerare).

SIMULAZIONE IN LINGUAGGIO ASSEMBLY DEI FLIP-FLOP

Il nostro primo problema, quando tentiamo di simulare un flip-flop 7474, è che non c'è nessun segnale di clock nell'insieme d'istruzioni di un microcalcolatore. Invece noi dobbiamo supporre che gli eventi siano innescati eseguendo un'appropriata istruzione piuttosto che una transizione del segnale di clock.

Come rappresenteremo le uscite Q e $\overline{Q}$? Si potrebbero usare due bit di memoria per rappresentare queste due uscite:

Poichè noi stiamo trattando dati e non segnali, $\overline{Q}$ è ridondante. Perciò il singolo flip-flop si riduce ad un bit di memoria. Un dispositivo 7474, che contiene due flip-flop si riduce a due bit di memoria, uno per ogni flip-flop implementato sul chip. Non esiste alcuna sorpresa per questa conclusione. Ogni bit della memoria di lettura e scrittura di un microcalcolatore è un elemento bistabile semplice: potrebbe, in realtà, essere un flip-flop.

La logica di un flip-flop 7474 può essere rappresentata da istruzioni che azzerano un bit di memoria, lo posizionano a 1 o vi memorizzano un bit non noto.

Supponiamo che i bit di memoria siano assegnati come segue:

La tabella di funzionamento del 7474 diventa, ora, queste istruzioni:

Preset	Clear	D	Primo flip-flop	Secondo flip-flop

L
H
	H
	H
		X
		H
			LD A,(FLP)
			SET 0,A
			LD (FLP),A
				LD A,(FLP)
				SET 1,A
				LD (FLP),A

H
H
	L
	H
		X
		L
			LD A,(FLP)
			RES 0,A
			LD (FLP),A
				LD A,(FLP)
				RES 1,A
				LD (FLP),A

L	L	X	Non si applica	

Riguardo alla tabella precedente, si presume che i bit 0 e 1 della parola di memoria identificata da FLP siano equivalenti al dispositivo a due flip-flop 7474. Le istruzioni LD spostano la parola tra memoria ed Accumulatore. Nell'Accumulatore l'istruzione SET posiziona ad 1 il bit appropriato; l'istruzione RES posiziona il bit specificato a zero.

SIMULAZIONE CON MICROCALCOLATORE DEI FLIP-FLOP IN GENERALE

In conclusione, un flip-flop diventa un singolo bit di memoria a lettura e scrittura in un sistema a microcalcolatore.

In un sistema a microcalcolatore tutti i flip-flop sono uguali. La logica dei flip-flop si riduce a queste quattro domande:

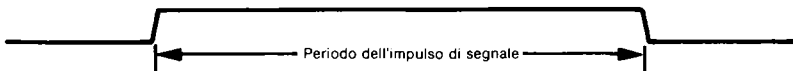
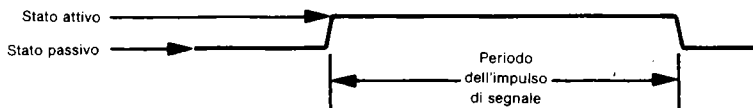
1) Quando si esegue un'istruzione per posizionare un bit di memoria ad 1?
2) Quando si esegue un'istruzione per posizionare un bit di memoria a 0?
3) Quando si esegue un'istruzione per memorizzare un bit in un bit di memoria?
4) Quando si esegue un'istruzione per leggere il contenuto di un bit di memoria?

SIMULAZIONE CON MICROCALCOLATORE DI DISPOSITIVI IN TEMPO REALE

Ci sono due tipi di dispositivi in tempo reale che considereremo: il monostabile (compresi i multivibratori monostabili) e il flip-flop master-slave. Specificatamente si descriveranno questi dispositivi:

- Il multivibratore monostabile 555 Signetics.
- Il multivibratore monostabile 74121.
- Il doppio flip-flop J-K master-slave con Clear 74107.

MONOSTABILE

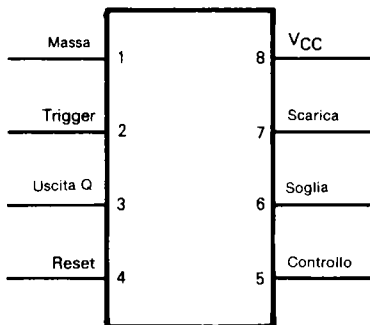
Un monostabile è un dispositivo che genera un impulso di segnale con un periodo di tempo specificato:

Un multivibratore monostabile è un dispositivo con un solo stato stabile o passivo. Esso produce segnali di uscita di monostabili come illustrato sopra, il cui impulso è uno stato instabile o attivo:

Il dispositivo è un "multivibratore" perchè può fare uscire una stringa continua di segnali — molto simile ad un segnale di clock. In altre parole, un'uscita di un multivibratore consiste in una stringa continua di segnali di monostabili.

Il periodo di tempo dell'impulso di segnale è un valore in tempo reale — è un numero finito di microsecondi o millisecondi o anche secondi.

FLIP-FLOP MASTER-SLAVE

Un flip-flop master-slave è un flip-flop che genera segnali di uscita basati sulla condizione dei segnali di ingresso qualche istante prima. Incontreremo nuovamente un valore in tempo reale — il ritardo tra ingressi ed uscite.

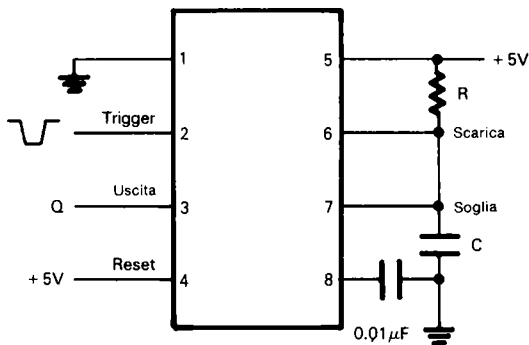
IL MULTIVIBRATORE MONOSTABILE 555

Il multivibratore monostabile 555 Signetics si può illustrare come segue:

Il fronte negativo di un segnale di clock sull'ingresso Trigger (pin 2) provoca una transizione da negativa a positiva dell'uscita Q. La durata dell'uscita Q a livello alto è controllata da un circuito resistenza/condensatore collegato ai pin di Scarica e di Soglia (7 e 6 rispettivamente).

Il Reset è un ingresso di azzeramento standard; un ingresso basso manterrà bassa l'uscita Q.

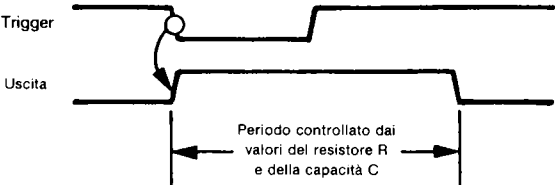
Il pin di controllo è usato per controllare la tensione nel multivibratore; non è significativo per una comprensione completa di come funziona il dispositivo 555.

I pin di terra e di alimentazione (1 e 8 rispettivamente) si spiegano da soli.

Ecco una maniera per configurare il multivibratore monostabile 555:

Non appena il livello di segnale varia da alto a basso, all'ingresso di Trigger il condensatore tra il pin 6 e la terra si carica. I livelli di segnale ai pin di Soglia e di Scarica, controllati dalla resistenza R e dal condensatore C, controllano il periodo per il quale Q sarà alto. Questo periodo di tempo è dato dalla seguente equazione:

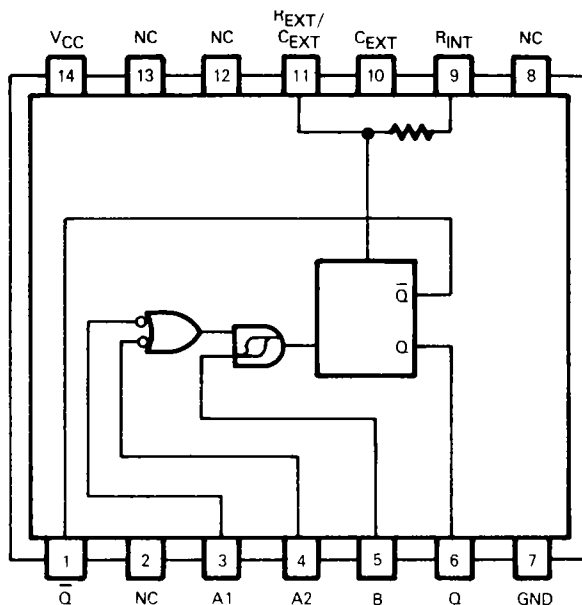
$$T = 1{,}1\ RC$$

Dove: T è il tempo in secondi
R è la resistenza in megaohm
C è la capacità in microfarad

Un impulso di segnale in uscita è generato come segue:

IL MULTIVIBRATORE MONOSTABILE 74121

Il multivibratore monostabile 74121 si può illustrare come segue:

TABELLA DI FUNZIONAMENTO

INGRESSI			USCITE		
A1	A2	B	Q	$\overline{Q}$	
L	X	H	L	H	Uscite monostabili
X	L	H	L	H	
X	X	L	L	H	
H	H	X	L	H	
H	↓	H	⊓	⊔	Uscite impulsive (one-shot)
↓	H	H	⊓	⊔	
↓	↓	H	⊓	⊔	
L	X	↑	⊓	⊔	
X	L	↑	⊓	⊔	

Un ingresso basso costante su A1, A2 o B manterrà il multivibratore monostabile 74121 nella sua condizione stabile — con un'uscita Q bassa e un'uscita Q alta. Gli ingressi alti su A1 e A2 hanno lo stesso effetto.

Ci sono cinque combinazioni di segnali in ingresso che genereranno uscite monostabili, Queste combinazioni di segnali in ingresso sono identificate nella tabella di funzionamento precedente.

Con riguardo alla tabella di funzionamento, i simboli sono usati come segue:

X rappresenta un "non prestare attenzione" (don't care)

↓ rappresenta una transizione logica da uno a zero

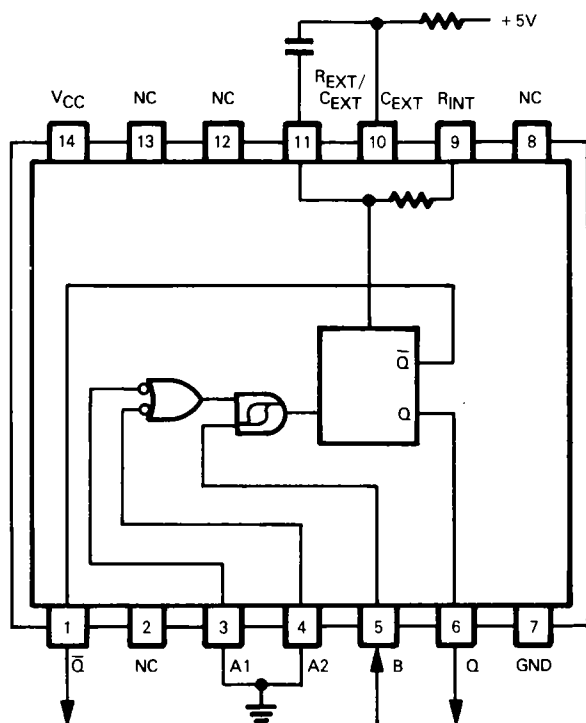
↑ rappresenta una transizione da zero a uno

⊓ rappresenta un impulso con un livello logico zero di monostabile e un livello uno di impulso

⊔ è la negazione (NOT) di ⊓

La durata dell'uscita impulsiva è determinata da una rete resistenza — capacità, proprio come descritto per il multivibratore monostabile 555 Signetics; ma ci sono alcune differenze. Il 74121 fornisce una resistenza interna alla quale si può accedere collegando R_{INT} (pin 9) a V_{CC} (pin 14). Una resistenza esterna variabile può essere collegata tra R_{INT} (pin 9) o R_{EXT} (pin 11) e V_{CC} (pin 14).

Una capacità di temporizzazione esterna, se presente, sarà collegata tra C_{EXT} (pin 10) e R_{EXT} (pin 11).

Ecco un modo in cui si può collegare un multivibratore monostabile 74121:

L'uso del multivibratore monostabile 74121 corrisponde alle due linee inferiori della tabella di funzionamento.

Una rete esterna di resistenza – capacità controlla la durata dell'impulso del monostabile. Ogni impulso di monostabile sarà innescato da una transizione basso alto sul pin 5 (B).

Dal punto di vista della programmazione, ci sono solo due aspetti significativi del multivibratore monostabile 74121:

1) **Le uscite del monostabile sono equivalenti a bit di valore fissato.** Una qualsiasi istruzione immediata che carica uno zero o un uno in un registro simula l'uscita del monistabile. Ecco un esempio:

   LD B,4 Poni ad 1 il bit 3 del Registro B, azzera tutti gli altri bit.

   Il bit 3 del Registro B è equivalente ad un flip-flop; così ogni altro bit del Registro B e di tutti gli altri registri.

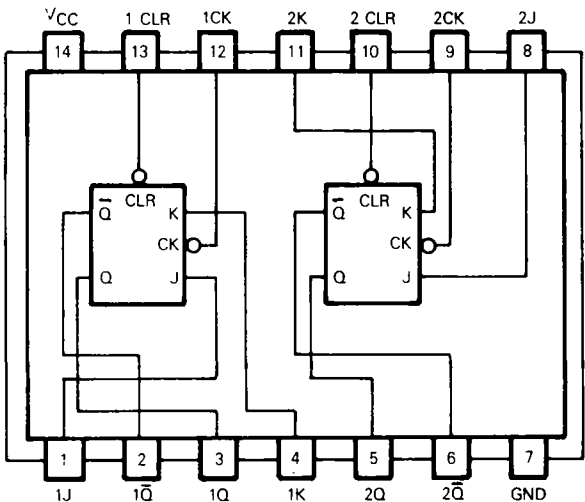
2) **Un'uscita impulsiva diventa un ritardo di tempo di valore fissato.** Mostreremo come si può calcolare questo ritardo di tempo in un sistema con microcalcolatore, ma prima esaminiamo il flip-flop master-slave 74107.

IL DOPPIO FLIP-FLOP MASTER-SLAVE J-K CON CLEAR 74107

Consideriamo il flip-flop master-slave. Questo flip-flop è illustrato come segue:

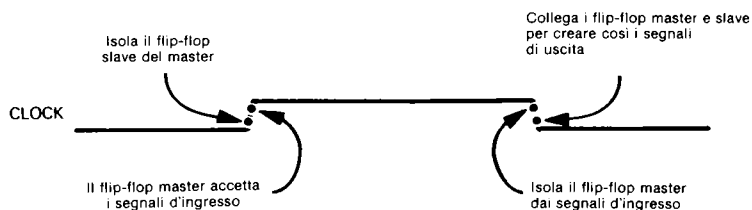
INGRESSI				USCITE	
1CLR o 2CLR	1CK o 2CK	1Jo 2J	1K o 2K	1Q o 2Q	1Q o 2$\overline{Q}$
L	X	X	X	L	H
H	⎍	L	L	Rimane nello stato precedente	
H	⎍	H	L	H	L
H	⎍	L	H	L	H
H	⎍	H	H	Cambia stato senza alcun riguardo dello stato precedente	

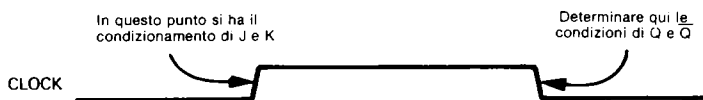
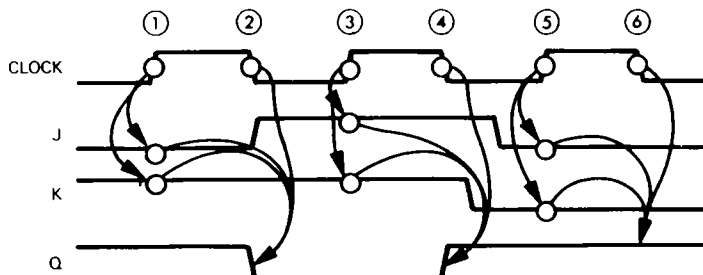
⎍ identifica un impulso di clock; il modo nel quale esso è usato è descritto sotto

X significa "non importa"

Esaminiamo la tabella di funzionamento illustrata sopra. Se non si ha familiarità con questo tipo di dispositivo logico, le sue caratteristiche non si evidenzieranno da sole.

FLIP-FLOP MASTER-SLAVE

L'annotazione "master-slave" identifica un circuito che è praticamente formato da due flip-flop. Perciò ci sono quattro flip-flop nel dispositivo 74107 illustrato sopra.

I flip-flop in ogni coppia master-slave rispondono ad un segnale di clock come segue:

Il significato di questa risposta al segnale di clock è che gli ingressi del flip-flop devono essere presenti sul fronte positivo del segnale di clock; questi ingressi devono rimanere stabili mentre il segnale di clock è alto. Le uscite del flip-flop tuttavia non cambieranno stato fino al fronte negativo del segnale di clock.

Il segnale di clock può essere usato per creare ritardi di tempo. L'uscita del flip-flop 74107 è determinata dai livelli che i segnali di ingresso hanno qualche periodo di tempo precedente. Ciò può essere illustrato come segue:

Ecco un esempio specifico:

La descrizione seguente del diagramma di temporizzazione illustrato sopra fa riferimento ai numeri cerchiati sopra il segnale di clock.

In ② , l'uscita Q va alta, perchè in ① J va basso e K va alto.
In ④ , Q cambia stato, perchè in ③ J e K erano entrambi alti.
In ⑥ , Q rimane inalterato, perchè ⑤ J e K erano entrambi bassi.

SIMULAZIONE DEL TEMPO REALE CON MICROCALCOLATORE

Qual'è l'aspetto significativo del multivibratore monostabile 555 e dei flip-flop master-slave? Quando si fa la simulazione con microcalcolatore di questi dispositivi, c'è un solo aspetto che è importante per la nostra discussione presente — e cioè il concetto di tempo reale.

Il multivibratore monostabile 555 crea impulsi di livello logico alto alle sue uscite, con durata di livello logico alto che è una funzione in tempo reale controllabile.

Il flip-flop master-slave 74107 permette la generazione di un segnale di uscita basato sulle condizioni di ingresso che esistono qualche istante di tempo precedente.

LOOP DI ISTRUZIONI DI TEMPORIZZAZIONE CON MICROCALCOLATORE

TEMPORIZZAZIONI DI INTERVALLI DI TEMPO BREVI

E' abbastanza semplice creare un ritardo di tempo usando un sistema con microcalcolatore — l'utilizzo di un sistema con microcalcolatore non significa che le operazioni vengano effettuate simultaneamente.
Consideriamo la seguente sequenza di istruzioni:

```
Cicli
           LD    A,TIME     ; Carica la costante di tempo nell'Accumulatore
  4 LOOP:  DEC   A          ; Decrementa l'Accumulatore
 10        JP    NZ, LOOP
                            ; Ridecrementa se non è zero
```

La sequenza di istruzioni precedente carica il valore di un dato, rappresentato dalla label TIME, nell'Accumulatore. L'Accumulatore è decrementato fino a zero, dopo di che l'esecuzione del programma continua. Supponiamo che si usi nel sistema del microcalcolatore un clock di 500 nanosecondi. Le istruzioni DEC e JP, considerate insieme, vengono eseguite in 14 cicli — equivalenti a 7 microsecondi. Ciò significa che la sequenza di programma illustrata sopra può provocare un ritardo di un valore minimo di sette microsecondi (quando TIME è uguale ad 1), ritardo che aumenta con incrementi di sette microsecondi fino ad un massimo di 1792 microsecondi, equivalenti a 7 x 256. Questo ritardo di tempo massimo si ha quando TIME ha un valore iniziale uguale a zero, poichè TIME è decrementato prima che sia controllato per vedere se è zero; perciò la fine del tempo avviene quando si decrementa 1 a 0, non quando si decrementa 0 a FF_{16}.

TEMPORIZZAZIONI DI INTERVALLI DI TEMPO LUNGHI

Ritardi di tempo più lunghi si possono generare disponendo di un contatore a 16 bit. Ecco la sequenza di istruzioni appropriata:

```
Cicli
           LD    DE,T16     ; Carica la costante di tempo in D ed E
  6 LOOP:  DEC   DE         ; Decrementa DE
  4        LD    A,D        ; Prova dello zero facendo l'OR del
  4        OR    E          ; Contenuto di D ed E con l'Accumulatore
 12        JP    NZ,LOOP    ;
```

La prima istruzione LD carica un valore di 16 bit, rappresentato dalla label T16, nella coppia di registri DE. L'istruzione LD, essendo un'istruzione immediata, crea tre byte di codice oggetto. Quando si esegue l'istruzione LD ecco che cosa succede:

VERIFICA DELLO STATO USANDO L'ISTRUZIONE DEC

L'istruzione DEC decrementa il valore a 16 bit dei registri DE come se fossa una sola entità di dato. Tuttavia, l'insieme d'istruzioni dello Z80 ha come peculiarità di trascurare il posizionamento dei bit di stato che si basano sul risultato del decremento di 16 bit. Ciò significa che non abbiamo nessun modo immediato di conoscere se i registri DE contengano ora un valore nullo o non nullo. Per provare ciò, carichiamo il contenuto del registro D nell'Accumulatore, quindi ne facciamo l'OR col contenuto del registro E. Se il risultato nell'Accumulatore è 0, allora entrambi i registri D ed E devono contenere 0. Se il risultato non è zero, si ritorna a decrementare il valore di 16 bit.

Si osservi che, per completare il loop d'istruzioni a tempi lunghi, si richiedono 26 cicli. Supponendo ancora che il microcalcolatore sia pilotato da un clock di 500 nanosecondi, occorreranno 13 microsecondi per eseguire una sola volta il loop d'istruzioni. Il valore minimo che T16 può avere è 1. Il massimo valore è ancora 0 perchè avviene un decremento prima del test sullo zero; inizialmente si potrebbe caricare 0 in D ed E, per cui sarebbe decrementato a $FFFF_{16}$ prima che si faccia il primo test sullo 0. In tale modo il loop d'istruzioni a tempi lunghi genererà ritardi, che variano con incrementi di 13 microsecondi, da un minimo di 13 microsecondi ad un massimo di 0,851968 secondi.

$$FFFF_{16} = 65535_{10}$$
$$13 \times 65536 = 851968 \text{ microsecondi}$$

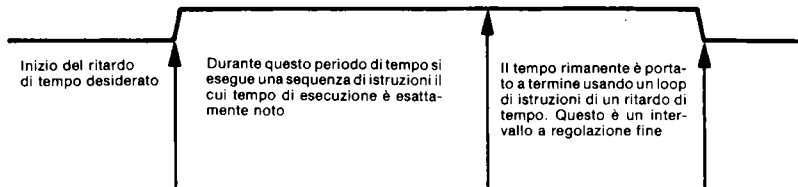
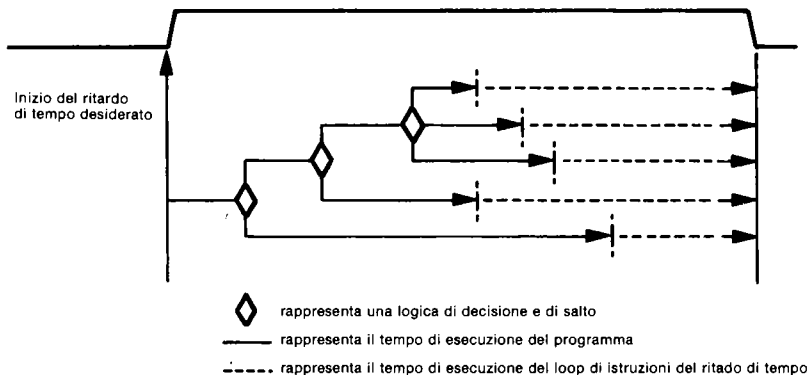
INIZIALIZZAZIONE DEL RITARDO DI TEMPO

Ora l'effettiva simulazione di un monostabile è complicata dal fatto che noi possiamo calcolare i ritardi di tempo, ma quando inizia il ritardo di tempo? Per dispositivi a logica digitale la risposta è semplice — il ritardo di tempo comincia quando un segnale d'ingresso cambia stato:

Per riferire questo concetto ad un programma di microcalcolatore, dobbiamo cominciare un ritardo di tempo al momento di completare qualche altre esecuzione di una sequenza di programmi. Questo concetto si può illustrare come segue:

```
        —
        —
        —
        JP      DELAY     ; Ultima istruzione di una sequenza precedente
        —
        —
        —
DELAY:  LD      A,TIME    ; Sequenza di istruzioni di un intervallo di tempo
LOOP:   DEC     A         ; breve
        JR      Z.LOOP
```

ESECUZIONE DI PROGRAMMI ENTRO RITARDI DI TEMPO

C'è un altro problema associato alla creazione di ritardi di tempo in un sistema con microcalcolatore, eseguendo loop d'istruzioni come abbiamo descritto: il microcalcolatore, in realtà non svolge nessun lavoro utile durante il ritardo di tempo. Ci può essere un semplice rimedio a questo problema stabilendo di poter definire un programma per il microcalcolatore da eseguire durante il periodo del ritardo di tempo. Ciò può essere illustrato come segue:

Dobbiamo supporre che possiamo calcolare il tempo esatto che ci vorrà per eseguire il nostro programma nel ritardo di tempo del monostabile; inoltre il tempo calcolato deve essere minore o uguale al ritardo di tempo. Non molti programmi si adatteranno a questa descrizione. Se, per esempio, si può fare eseguire più di una sequenza di istruzioni dipendenti da condizioni presenti, allora ci possono essere diversi tempi richiesti per eseguire un programma. Inoltre, dal momento che c'è un numero fissato di rami identificabili, il problema è trattabile e può essere illustrato come segue:

Ora ogni "derivazione" dei rami del programma finirà come segue:

```
LD    A,DLY1    ; Carica il primo ritardo di tempo
JP    LOOP      ; Inizio della loop del ritardo di tempo
—
—
—
LD    A,DLY2    ; Carica il secondo ritardo di tempo
JP    LOOP      ; Inizio della loop del ritardo di tempo
```

```
          —
          —
          —
          LD      A,DLY3    ; Carica il terzo ritardo di tempo
          JP      LOOP      ; Inizio della loop del ritardo di tempo
          —
          —
          —
          LD      A,DLY4    ; Carica il quarto ritardo di tempo
          JP      LOOP      ; Inizio della loop del ritardo di tempo
          —
          —
          —
          LD      A,DLY5    ; Carica il quinto ritardo di tempo
          JP      LOOP      ; Inizio della loop del ritardo di tempo
          —
          —
          —
LOOP:     DEC     A         ; Sequenza di istruzioni d'intervallo di tempo breve
          JR      NZ,LOOP   ; breve
```

E' molto comune che un programma di microcalcolatore contenga numerosi rami condizionali; ci possono essere centinaia di differenti tempi d'esecuzione, dipendenti dalle varie combinazioni delle condizioni correnti. L'esecuzione di un programma nell'intervallo di tempo del richiesto ritardo ora diventa non pratica, poichè la logica necessaria per calcolare il tempo rimanente per gli innumerevoli rami del programma è davvero troppo complicata.

I LIMITI DELLA SIMULAZIONE LOGICA

Un microcalcolatore Z80 può calcolare ritardi di tempo dal momento che nessun altro programma necessiti di essere eseguito durante il ritardo di tempo, o stabilendo una sequenza d'istruzioni molto semplice con diramazioni molto limitate da eseguire durante il ritardo di tempo.

RITARDI DI TEMPO SIMULTANEI

Non possiamo simulare contemporaneamente ritardi di tempo, nè possiamo simulare un ritardo di tempo che deve avvenire in parallelo alle esecuzioni di programmi paralleli non definibili. Questi ritardi devono essere gestiti da una logica esterna.

INTERFACCIAMENTO CON MONOSTABILI ESTERNI

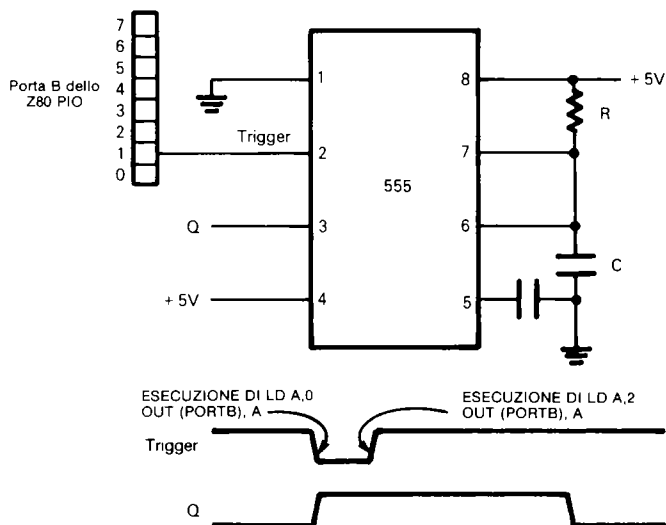
Notate che, anche se potete avere logica esterna per creare ritardi di tempo, è molto facile per il sistema con microcalcolatore dare l'avvio al ritardo di tempo e per la logica esterna riportare il completamento del ritardo di tempo.

INIZIALIZZAZIONE DELL'IMPULSO

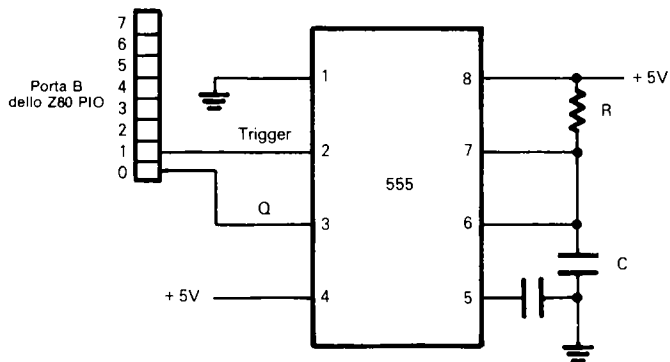
Possiamo identificare l'inizio di un ritardo di tempo facendo uscire semplicemente un bit appropriato. Riguardiamo il modo di uscita verso una logica esterna di una simulazione dell'invertitore di segnale. Il modo di far uscire un segnale verso una logica esterna è in realtà molto facile. Consideriamo le quattro seguenti istruzioni:

```
LD      A,0          ; Carica 0 nell'accumulatore
OUT     (PORT B),A   ; Uscita per mezzo della porta B di I/O
LD      A,2          ; Carica 1 nel bit 1 dell'accumulatore
OUT     (PORT B),A   ; Uscita per mezzo della porta B di I/O
```

Un 1 è fatto uscire sul pin 1 della porta B di I/O. Supponendo che il pin associato con questa porta di I/O sia collegato al trigger di un multivibratore e che questi fosse prima alto, allora la semplice esecuzione delle istruzioni precedenti innescheranno un impulso di monostabile.

Ciò può essere illustrato come segue:

E' ugualmente facile per la logica esterna segnalare la fine di un ritardo di tempo.

FINE DELL'IMPULSO USANDO LO STATO

Se ci stiamo occupando di logica "più grande che o uguale a", tutto ciò che è necessario per un'uscita di un monostabile è di essere collegata ad un altro pin di porta I/O del microcalcolatore.

I segnali che arrivano ai pin delle porte di I/O sono immagazzinati. Il programma che deve essere eseguito dal microcalcolatore può, in ogni istante, incamerare il contenuto delle porte di I/O e provare la condizione del bit 0, che è stato collegato all'uscita Q. Quando si trova che questo bit è 0, la logica del programma del microcalcolatore riconosce che è stato superato l'intervallo di tempo.

La seguente sequenza d'istruzioni proverà la porta di I/O e azzererà lo stato di "intervallo di tempo completo" che è in relazione al pin 0 della porta B di I/O:

```
        IN      A,(PORT B)    ; Fa entrare il contenuto della porta B di I/O nell'ac-
                              ; cumulatore
        BIT     0,A           ; Verifica il bit 0
        JP      NZ,NEXT       ; Continua se bit è 1
; Il programma di "time out" comincia qui
        –
        –
        –
NEXT:                         ; Il programma "time hot out" comincia qui
```

L'istruzione IN sposta il contenuto corrente della Porta B di I/O nell'accumulatore.

L'istruzione seguente BIT controlla il bit 0 dell'Accumulatore e posiziona nel modo seguente il flag Zero per rispecchiare il contenuto del bit:

Z	BIT
1	0
0	1

Se il bit d'ingresso del pin 0 della porta B di I/O è 1, allora l'uscita Q è ancora alta. L'istruzione JP NZ,NEXT fa semplicemente continuare l'esecuzione del programma.

Se il bit 0 della porta B di I/O è 0, allora il ritardo di tempo è finito; saltiamo ad una sequenza di programma che sarà eseguita solo immediatamente dopo un "time out".

INTERRUZIONI E TIME OUT

La fine esatta di un time out può essere segnalata al microcalcolatore usando un'interruzione.

Appena il colpo del monostabile finisce, esso forzerà il sistema microcalcolatore a interrompere l'esecuzione di qualunque programma fosse in fase corrente di esecuzione. Si forzerà un salto ad un altro programma che sarà stato specificatamente progettato per rispondere al "time out".

Le considerazioni della programmazione associate all'interruzione sono più complicate del livello di trattazione fatto nel Capitolo 2. Differiremo perciò una descrizione dettagliata del processo d'interruzione, successivamente in questo libro. Per il momento, è sufficiente capire che l'istante esatto di un "time out" può essere segnalato al sistema del microcalcolatore usando una logica d'Interruzione.

INTERFACCIAMENTO CON TEMPORIZZATORI PROGRAMMABILI

Si possono usare altri tipi di logica esterna per creare un circuito temporizzatore programmabile, quale lo Z80 CTC (Counter/Time Circuit). Il CTC è un dispositivo programmabile che contiene quattro circuiti contatori/temporizzatori programma-

bili con logica di controllo associata. Ogni contatore/temporizzatore può essere raggiunto dalla CPU come una porta di I/O o come una locazione di memoria.

Ognuno dei quattro contatori/temporizzatori può essere programmato per funzionare come un temporizzatore, ed è decrementato dal clock del sistema, o come contatore, e sarà decrementato dalla ricezione di un segnale di clock/trigger. Ci sono parecchie altre opzioni di funzionamento che possono essere stabilite sotto il controllo del programma — cioè scrivendo semplicemente una parola di controllo all'appropriato contatore/temporizzatore. Non tenteremo di descrivere qui tutte queste opzioni; lo Z80 CTC è descritto in dettaglio in An Introduction to Microcomputers: Volume II - Some Real Products. Guardiamo solo brevemente una sequenza tipica di eventi e la flessibilità e la semplicità ottenuta usando un temporizzatore con microcalcolatore.

Supponiamo che il CTC sia indirizzato come una porta di I/O — in realtà quattro porte di I/O, poichè ogni temporizzatore/contatore nel CTC funziona indipendentemente ed è scelto individualmente. Per iniziare un ritardo di tempo, effettueremo i passi seguenti:

1) Fare uscire una parola di controllo al desiderato contatore/temporizzatore, per specificare che esso deve funzionare come temporizzatore. La stessa parola di controllo specifica pure altre informazioni sui modi, quale la velocità di decremento del temporizzatore, quando il temporizzatore deve essere fatto partire, e così via.
2) Fare uscire una costante che rappresenta il ritardo di tempo desiderato verso il contatore/temporizzatore.

Appena si è fatta uscire la costante del ritardo di tempo, il temporizzatore comincerà a decrementarsi. Quando il conteggio sarà zero, si genererà un segnale di time out. Questo segnale può essere usato per informare la CPU che l'intervallo di tempo è completo. L'informazione potrebbe essere trasmessa usando un ingresso di interruzione della CPU o mediante qualche logica intermedia.

L'uso di un temporizzatore prgrammabile offre ovvi vantaggi rispetto ad un monostabile esterno. Il CTC può essere programmato e riprogrammato per fornire ogni ritardo di tempo desiderato, laddove un monostabile esterno può fornire un ritardo di tempo singolo e fisso. Inoltre il CTC fornisce quattro contatori/ temporizzatori così che possono generare ritardi di tempo contemporanei o sovrapponentesi.

Nell'esempio di progetto che sviluppiamo in questo libro si richiedono alcuni ritardi di tempo e non ci sono richieste di ritardi sovrapponentesi. Perciò useremo semplici loop di istruzioni di CPU per generare i ritardi richiesti. **Se la vostra applicazione richiede più di rozze sequenze di temporizzazioni, dovrete approfondire l'uso di temporizzatori programmabili.**

Capitolo 3

SIMULAZIONE DIRETTA DELLA LOGICA DIGITALE

I dispositivi logici discreti simulati nel Capitolo 2 non sono stati scelti a caso; messi correttamente in sequenza essi simuleranno la logica illustrata in Figura 3-1. Questa logica è una parte dell'interfaccia della stampante per le stampanti Serie Q - Qume e Serie Sprint. La Figura 3-2 è il diagramma della temporizzazione che si riferisce alla Figura 3-1. Descriveremo entrambi le figure ad un livello molto elementare.

Lo scopo di questo capitolo è di fornire una correlazione uno a uno tra programmazione in linguaggio assembly di un microcalcolatore e progetto logico digitale. Ciò che voi dovete capire è che tale correlazione uno a uno può essere forzata, ma non naturale — è qui dove si trova il problema di comprensione. Si potrebbero scrivere programmi su microcalcolatori per mettere in risalto la natura dei microcalcolatori, non le caratteristiche di logica digitale.

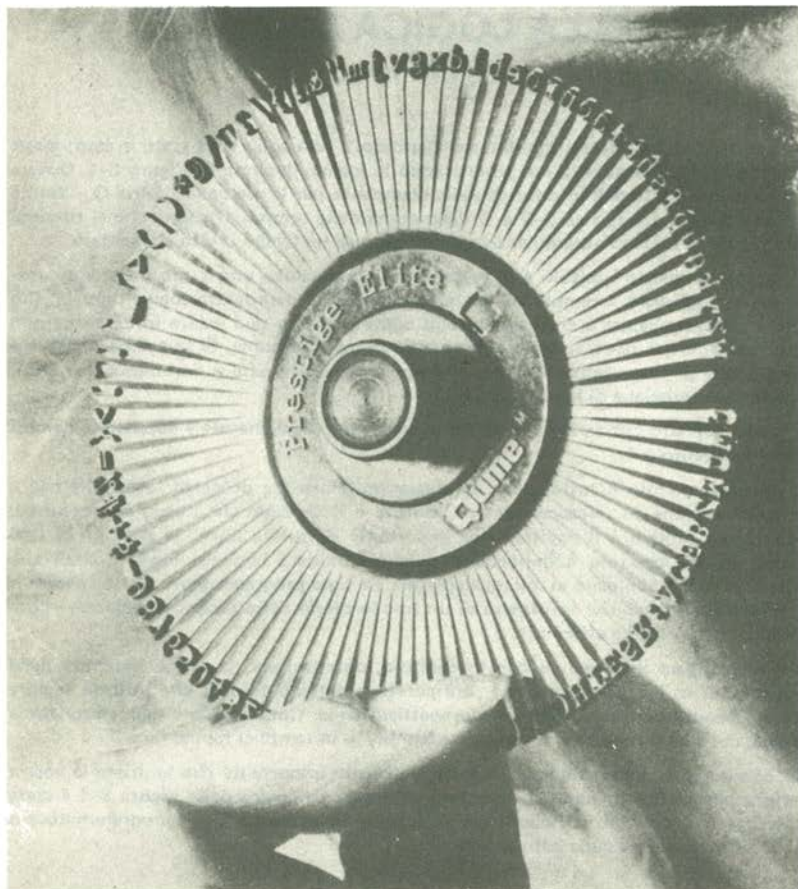
Il modo corretto per scrivere programmi su un microcalcolatore è descritto a cominciare dal Capitolo 4.

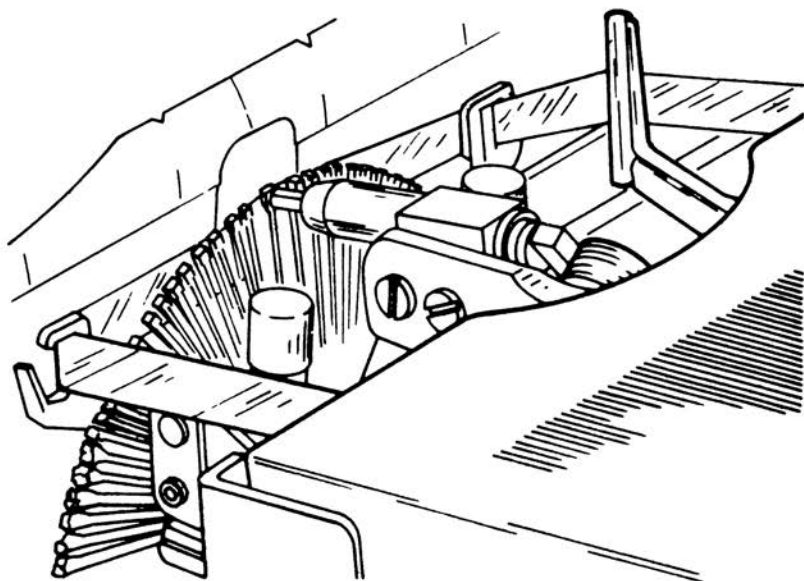
In questo capitolo è sottolineata la giustapposizione tra progetto logico digitale e programmazione su microcalcolatore. Questo è il capitolo che unisce due concetti; per questo motivo è il capitolo più importante di questo libro. Se siete progettisti logici, questo capitolo è importante poichè eliminerà concetti logici digitali che non si possono applicare ai microcalcolatori. Se siete programmatori, questo capitolo è importante perchè vi informerà di un nuovo scopo della programmazione — l'implementazione logica efficiente.

Per raggiungere lo scopo di questo capitolo, descriveremo la logica illustrata nelle Figure 3-1 e 3-2; la descrizione sarà curata e dettagliata così che potrete seguire questo capitolo anche se non siete progettisti logici. Col procedere della descrizione logica, ci addentreremo nel linguaggio assembly — in semplici tappe.

Se conoscete la logica digitale, è particolarmente importante che limitiate la vostra lettura alle scritte in neretto di questo capitolo. La logica della Figura 3-1 è stata descritta con dettagli sufficienti per soddisfare le necessità di un programmatore o di un lettore senza alcuna base di logica.

COME FUNZIONA LA STAMPANTE QUME

L'elemento attivo di stampa Qume è una ruota di stampa a 96 petali, con un solo carattere su ogni petalo.

PER GENTILE CONCESSIONE DELLA QUME CORPORATION

Si stampa un carattere muovendo la ruota di stampa fino a che il petalo appropriato si trovi di fronte a un martelletto di stampa comandato da un solenoide. Si alimenta quindi il martelletto di stampa, che colpisce il petalo della ruota di stampa, che segna la carta:

Ogni volta che un carattere non deve essere stampato, la ruota di stampa è posizionata con un petalo corto immediatamente verticale così che il carattere appena stampato è visibile.

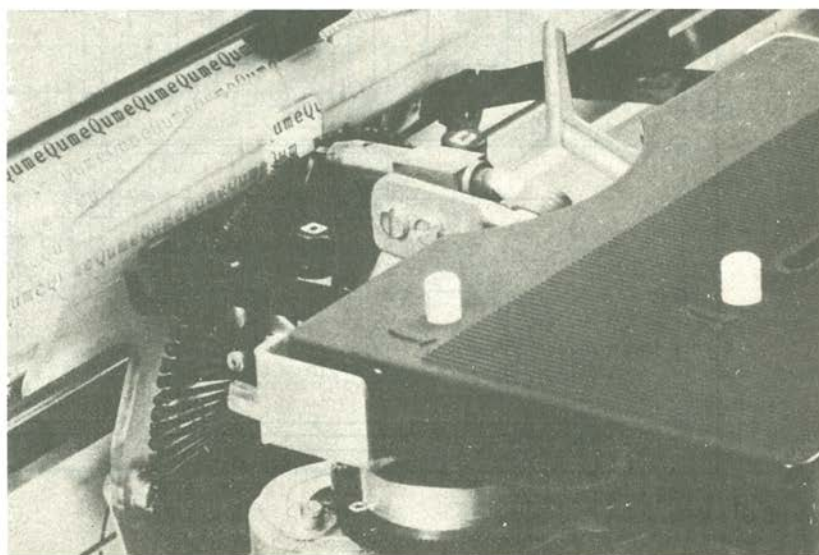
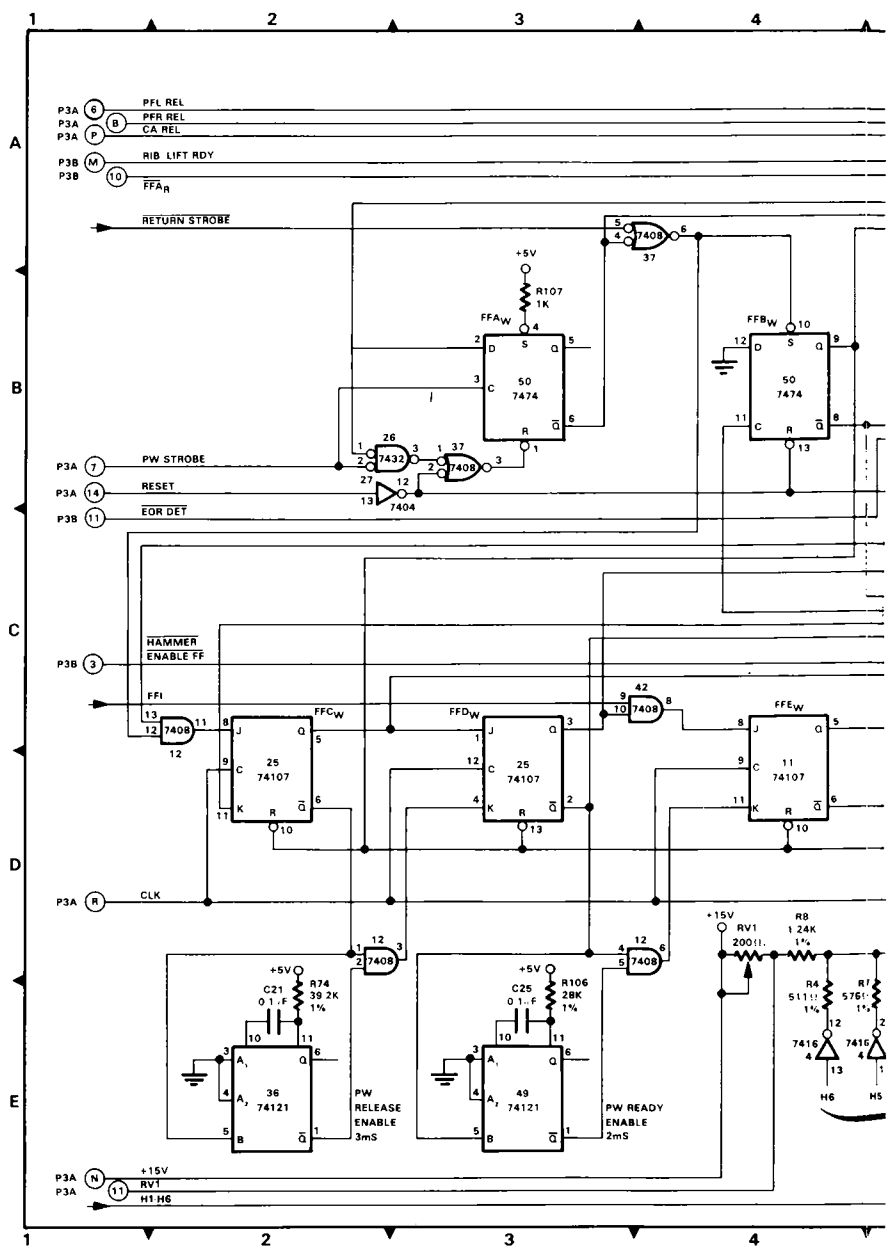



Figura 3-1. Logica di controllo della ruota di stampa

Figura 3-1. Logica di controllo della ruota di stampa

CH RDY
P3A
FFA_W
FFB_W
FFE_W
PW SETTLING
2 mSEC
74121
7411
FFF_W
FFD_W
PW REL
7432
7408
HAMMER DIS
HAMMER PULSE
7404
74107
START RIB MOTION PULSE
CLK
555
7416
H5
H4
H3
H2
H1

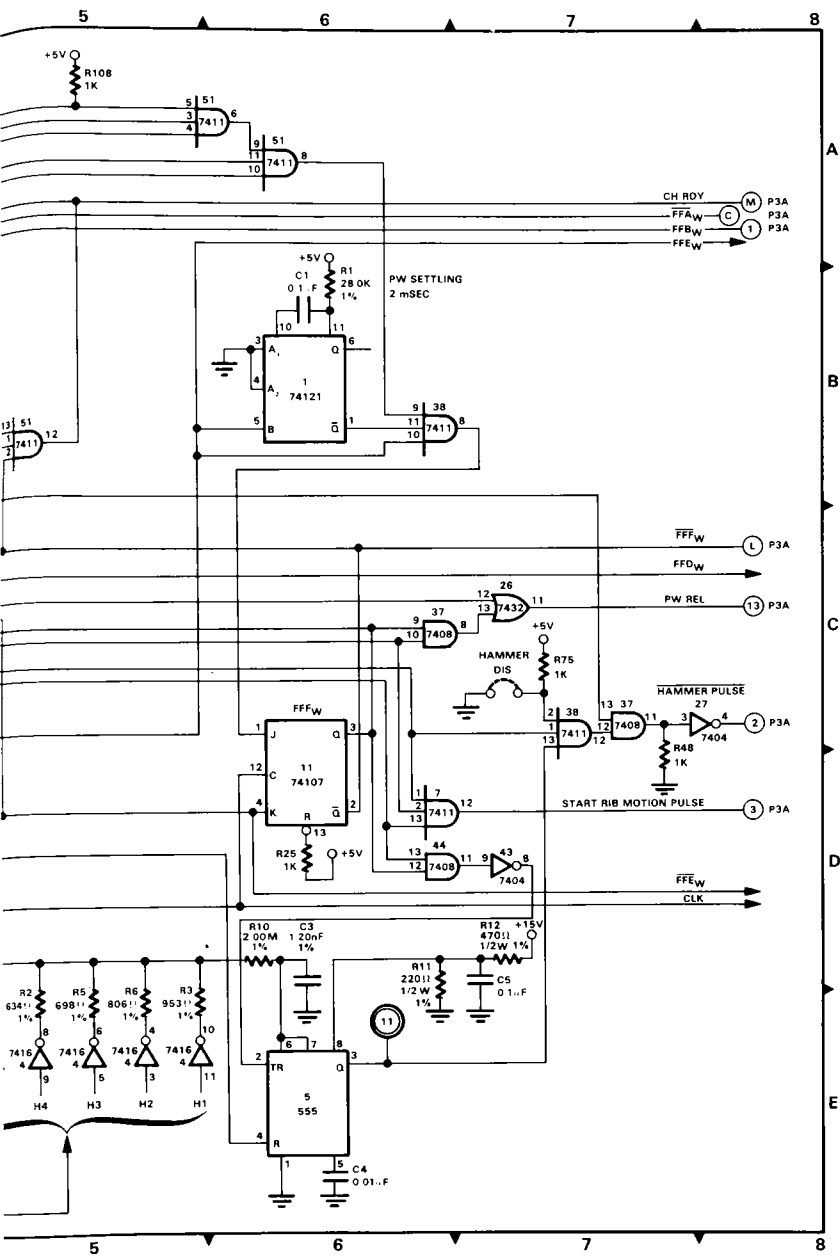
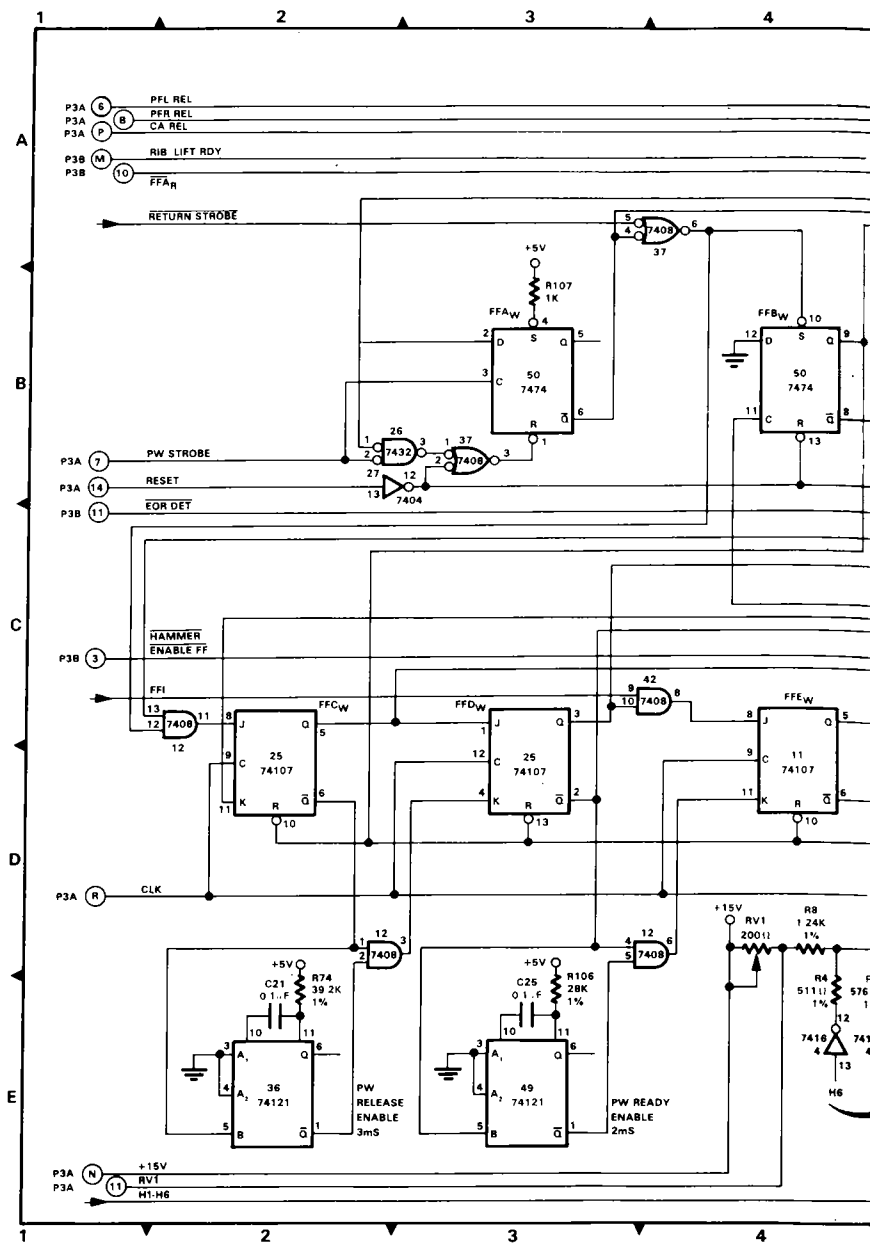
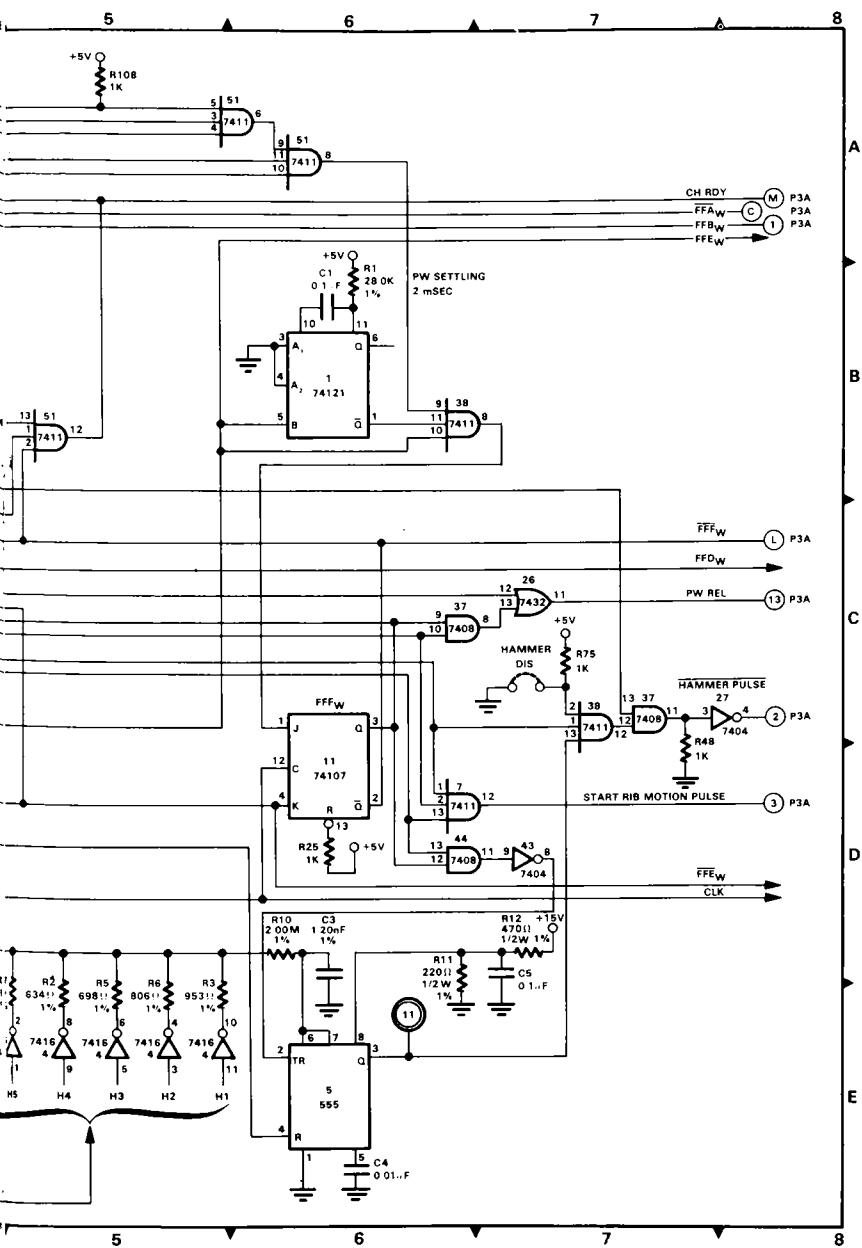
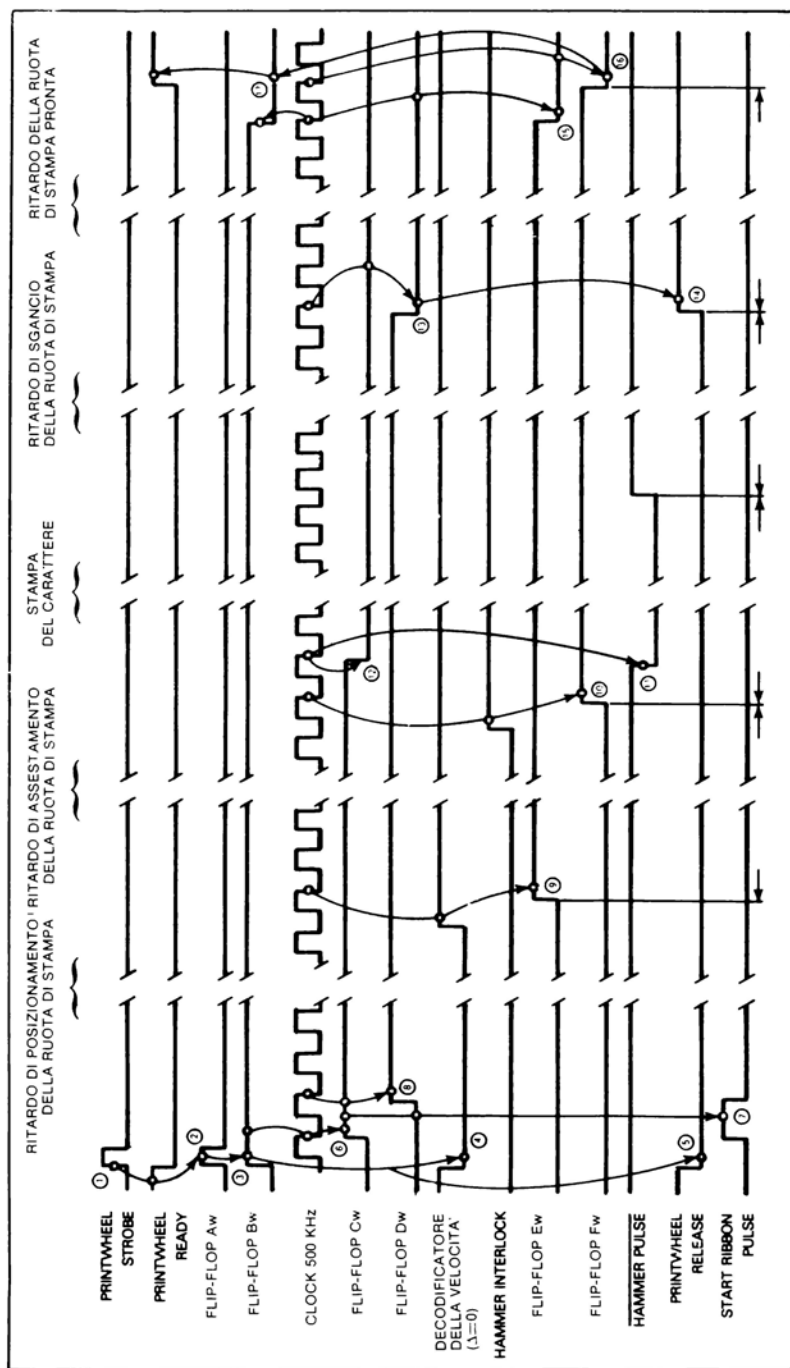
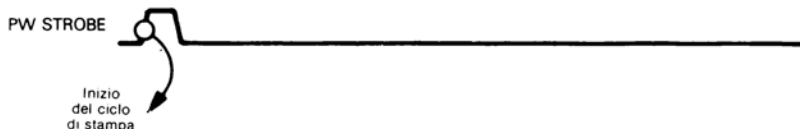



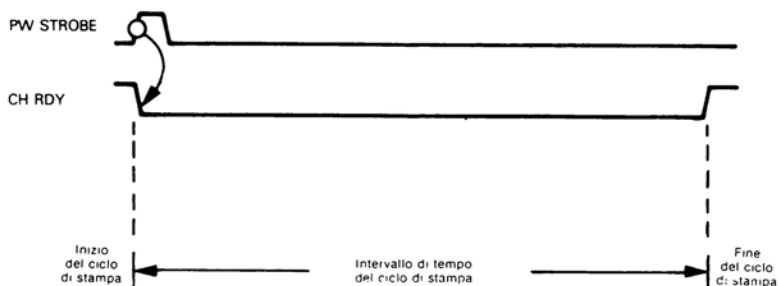
Figura 3-2. Diagramma della temporizzazione della logica di controllo della ruota di stampa

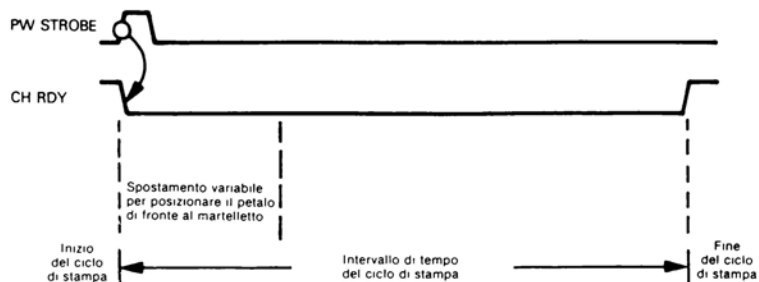
Come parte del ciclo di stampa, si deve muovere il nastro di stampa ed il carrello della carta.

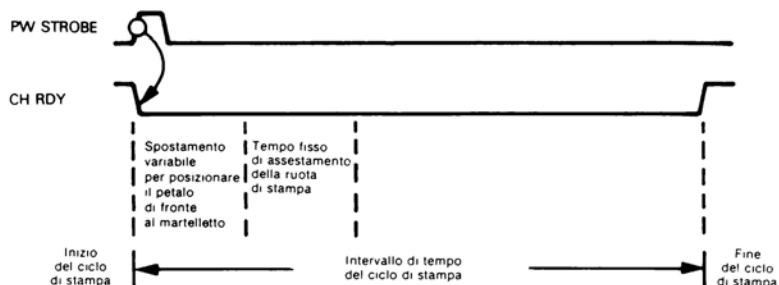
Ogni carattere è stampato in accordo ad una definita sequenza di eventi, ai quali ci si riferisce globalmente come "ciclo di stampa". La logica illustrata nella Figura 3-1 controlla il ciclo di stampa di un carattere. **In un ciclo di stampa devono accadere i seguenti eventi:**

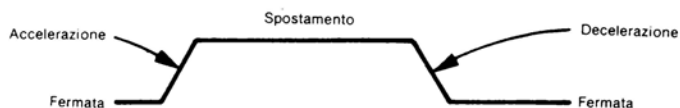
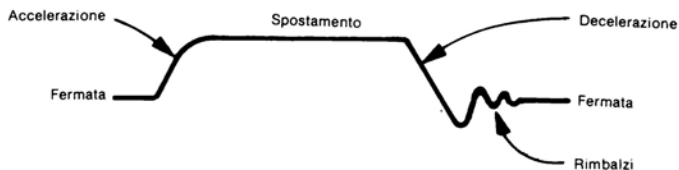
PW STROBE

1) Dapprima si deve far iniziare il ciclo di stampa. **Per iniziare il ciclo di stampa si dà un impulso alto ad un segnale (PW STROBE).**

PRINT WHEEL READY
CH RDY

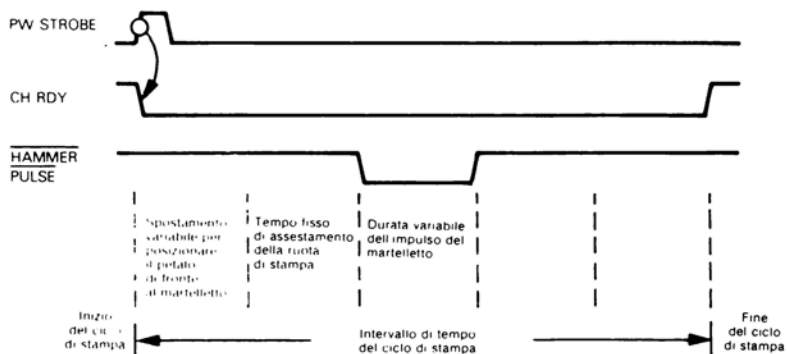
2) Il ciclo di stampa durerà per un intervallo di tempo fissato. Ovviamente, durante questo intervallo di tempo non deve iniziare nessun altro ciclo di stampa. Perciò, **si deve fornire alla logica esterna** responsabile della generazione del vero PW STROBE **un segnale che identifichi la durata del ciclo di stampa. Questo segnale è PRINTWHEEL, READY, chimato pure CH RDY:**

La sequenza di eventi che provoca realmente la stampa di un carattere può ora procedere, con la sicurezza che la logica esterna non tenterà di cominciare a stampare il prossimo carattere prima di completare il corrente ciclo di stampa.

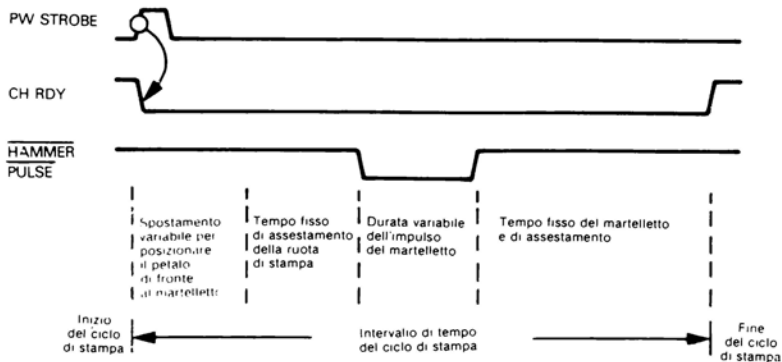
3) **La ruota di stampa si muove dalla sua posizione di visibilità finchè l'appropriato petalo del carattere non sia di fronte al martelletto di stampa:**

Occorre un ritardo di tempo variabile per la logica di posizionamento della ruota di stampa. Ovviamente occorrerà un tempo maggiore per posizionare un petalo lontano dalla posizione di visibilità che per posizionare un petalo adiacente.

4) Prima che si alimenti il martelletto **si deve dare un tempo di assestamento alla ruota di stampa.** E' sufficiente un ritardo di tempo fisso di due millisecondi:

RITARDI DI ASSESTAMENTO

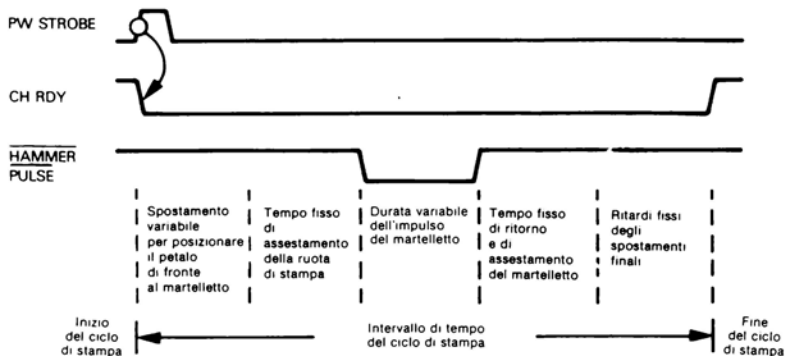
I ritardi di tempo di assestamento sono un aspetto molto importante della logica di supporti di un qualsiasi movimento meccanico. E' facile disegnare una linea che mostri la velocità del movimento, come segue:

Ma in realtà il movimento avviene così:

Il rimbalzo che segue la decelerazione deve essere parato con un ritardo di tempo di assestamento.

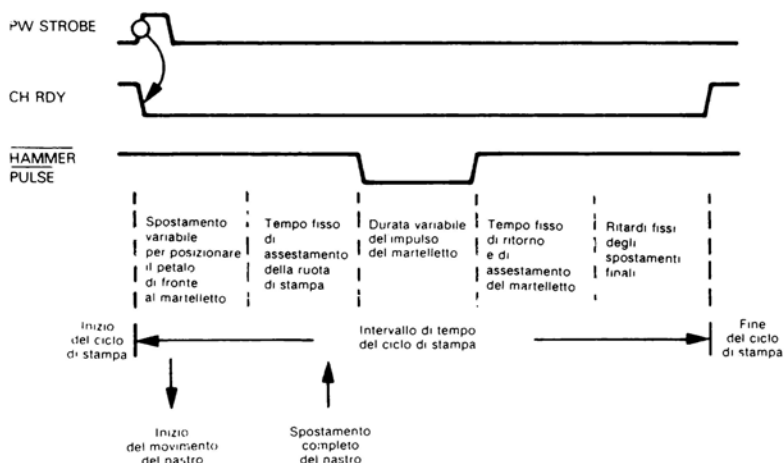
Si stamperà un carattere confuso se la ruota di stampa starà ancora vibrando quando il martelletto di stampa colpisce la carta con un petalo.

5) Alla fine del ritardo di tempo di assestamento, **si può alimentare il martelletto.** Ciò è ottenuto inviando un impulso ad un solenoide. **Si forniscono sei intensità per gli impulsi che alimentano il martelletto**, poichè alcuni caratteri hanno un'area superficiale più sostanziosa di altri. Per far battere un'area superficiale relativamente grande come un "W" con la stessa intensità per far battere un carattere piccolo, come ".", si produrrebbero diversità nel testo stampato.
La durata dell'impulso per il solenoide del martelletto di stampa è controllato dal seguente ritardo di tempo:

La barra sopra $\overline{\text{HAMMER PULSE}}$ identifica il segnale attivo quando è basso.

6) Quando il ritardo di tempo dell'impulso del martelletto di stampa è completato, il martelletto ha battuto un petalo e lo ha forzato sulla carta. Ora **si deve dare tempo al martelletto di tornare alla sua posizione prima dell'alimentazione.** A tale scopo si genera un ritardo di tempo di tre millisecondi:

POSIZIONE DI VISIBILITA' DELLA RUOTA DI STAMPA

7) Ora **si può mettere la ruota di stampa nella sua posizione di visibilità** e si può fare avanzare il carrello della carta nella posizione del prossimo carattere. La "posizione di visibilità" della ruota di stampa è la sua posizione normale di riposo. In questa posizione, un petalo corto è di fronte al martelletto di stampa, così il carattere stampato più recentemente è visibile sopra il petalo corto; da qui "posizione di visibilità". Non avendo dato tempo al martelletto di stampa di riposizionarsi indietro prima di muovere la ruota di stampa nella sua posizione di visibilità, si può interrompere il petalo della ruota durante la percussione della punta, con il martelletto che sta ancora spingendosi fuori. Inoltre, la carta potrebbe macchiarsi contro un petalo piegato. Perciò si deve dare tempo al martelletto di ritrarsi pienamente, per non causare questi problemi.

Un ritardo di tempo finale di due millisecondi permette alla ruota di stampa e al carrello della carta di riposizionarsi:

IMPULSO D'AVVIO DEL NASTRO
FFA

8) Che cosa si può dire sulla logica di controllo del nastro? **Per impressionare in modo pulito la carta, un pezzo di nastro fresco deve trovarsi tra il petalo del carattere e la carta.** Poco dopo l'inizio del ciclo di stampa, s'invia quindi un segnale (IMPULSO D'AVVIO DEL MOTO DEL NASTRO) alla logica esterna, segnale che controlla realmente il movimento del nastro. Questa logica esterna, (non fa parte della Figura 3-1) risponde con un segnale di movimento del nastro completato ($\overline{FFA}$), poichè non si può permettere che il martelletto di stampa venga sparato mentre il nastro è ancora in movimento. **In tale modo si fa avanzare il nastro mentre inizialmente si posiziona e si sistema la ruota di stampa:**

Riassumendo, un ciclo di stampa consiste in cinque ritardi di tempo; ogni ritardo di tempo prende avvio da un'eccitazione di una logica attiva seguita da un periodo di movimento meccanico.

SEGNALI DI INGRESSO E DI USCITA

Ora che avete una comprensione generale delle funzioni controllate dalla logica della Figura 3-1, **il prossimo passo sarà di guardare più da vicino i segnali d'ingresso e d'uscita.**

Per conoscere che cosa fare e quando farlo, dobbiamo fare pieno affidamento sui segnali d'ingresso. Analogamente i segnali d'uscita rappresentano il solo modo di trasmettere informazioni di controllo ad una logica esterna.

Il nostro limitato scopo, a questo punto, è di capire quale funzione svolge ogni ingresso ed ogni uscita, e come — fisicamente — intendiamo maneggiare i segnali. Discuteremo dapprima il "come".

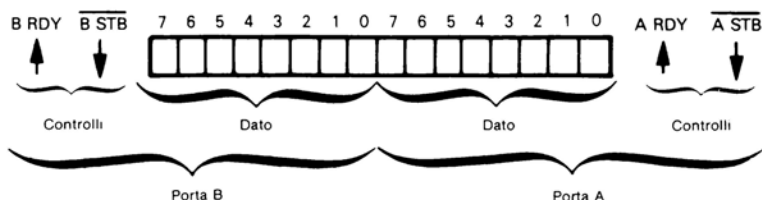
DISPOSITIVI D'INGRESSO/USCITA

INTERFACCIA PARALLELA DI INGRESSO/USCITA

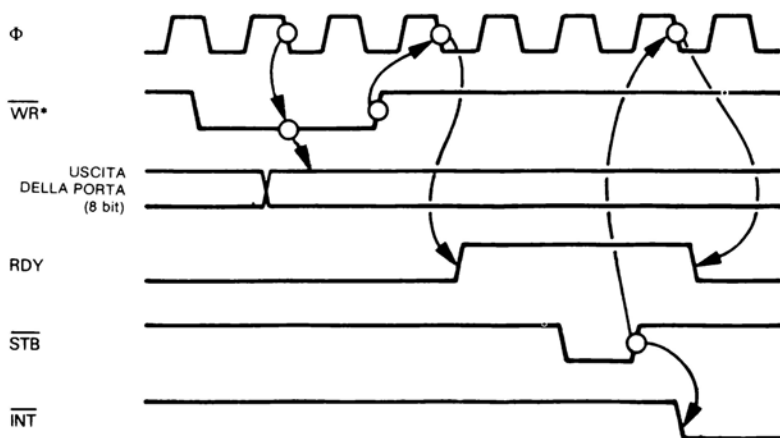
Il principale dispositivo usato per trasmettere segnali e dati tra un sistema con microcalcolatore Z80 e una logica esterna è l'Interfaccia Parallela d'Ingresso/Uscita Z80 (PIO). Useremo due dispositivi Z80 PIO.

Poichè questo dispositivo è stato descritto in An Introduction to Microcomputers, supporremo che voi conosciate superficialmente le sue capacità e la sua organizzazione; se non fosse così, guardate An Introduction to Microcomputers: Volume II – Some Real Products prima di continuare. Altrimenti non capirete la discussione che segue.

L'INTERFACCIA PARALLELA D'INGRESSO/USCITA Z80 (PIO)

L'interfaccia parallela d'ingresso/uscita Z80 (PIO) fornisce 16 pin d'I/O che possono essere raggruppati in porte d'I/O come segue:

Ogni porta ha due segnali di controllo associati, RDY e $\overline{\text{STB}}$, per usare trasferimenti di dati in parallelo con "handshaking" automatico.
RDY è inviato dallo Z80 PIO alla logica esterna; $\overline{\text{STB}}$ è inviato dalla logica esterna in ingresso dello Z80 PIO.

MODI DELLA PORTA DI I/O

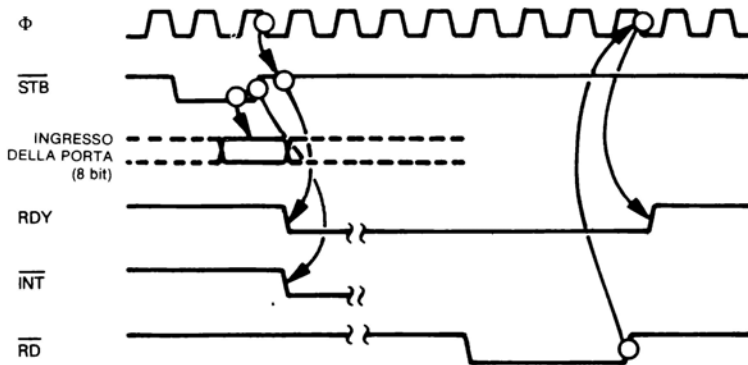
Ogni porta può essere programmata per funzionare in tre modi; inoltre, la Porta A può funzionare in un quarto modo che non è disponibile per la Porta B. La Porta A e la Porta B non devono necessariamente funzionare nello stesso modo.

Vediamo ora i modi dello Z80 PIO.

Il modo di Uscita (Modo 0) permette di usare la Porta A e/o la Porta B come condotto per trasferire dati alla logica esterna. L'"handshaking" funziona come illustrato in fondo a pagina 3-10.

Quando la CPU esegue una istruzione di uscita, genera dei segnali di controllo che lo Z80 PIO combina in un impulso interno di scrittura. Nel diagramma precedente ciò è indicato dal segnale $\overline{WR}$*. Φ è un clock di sistema che la logica dello Z80 PIO usa per sincronizzare le transizioni dei suoi segnali interni.

USCITA DALLO Z80 PIO CON "HANDSHAKING"

Quando la CPU esegue un'istruzione di uscita avente accesso a porte di I/O si inizia un ciclo di uscita. L'impulso di scrittura ($\overline{WR}$*) è usato per creare uno "strobe" del dato sul Bus dei Dati e nel registro d'uscita della porta di I/O indirizzata. Dopo l'impulso di scrittura, sulla successiva transizione alto — basso dell'impulso di clock Φ, si invia alla logica esterna il segnale di controllo RDY alto. RDY rimane alto finchè la logica esterna non invia un impulso basso sull'ingresso $\overline{STB}$. Sulla seguente transizione alto — basso dell'impulso di clock Φ, RDY torna basso. La transizione basso — alto di $\overline{STB}$ genera inoltre una richiesta di interruzione — se le interruzioni sono state abilitate.

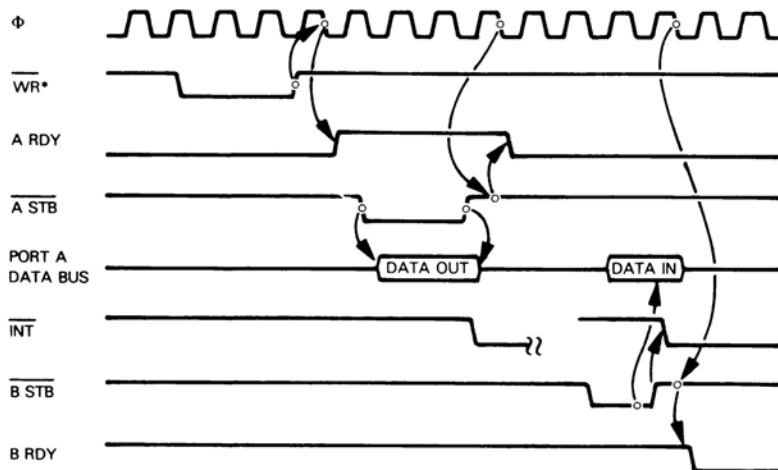
La temporizzazione per il modo di Ingresso (Modo 1) è illustrata sotto:

INGRESSO NELLO Z80 PIO CON "HANDSHAKING"

La logica esterna inizia in ciclo di ingresso con un impulso basso su $\overline{STB}$. Questo impulso basso fa si che lo Z80 PIO carichi i dati dai pin della porta di I/O nel registro della porta d'ingresso. Sul fronte di salita dell'impulso $\overline{STB}$ si potrà innescare una richiesta d'interruzione se saranno state abilitate le interruzioni.

Sul fronte di discesa dell'impulso di clock Φ che segue l'ingresso $\overline{STB}$ alto, si fa uscire RDY basso per informare la logica esterna che il suo dato è stato ricevuto ma che non è ancora stato letto. RDY rimarrà basso finchè la CPU non avrà letto il dato, istante nel quale RDY sarà riportato alto.

E' compito della logica esterna assicurarsi che il dato non sia fatto entrare nello Z80 PIO mentre RDY è basso. Se la logica esterna mette dati in ingresso allo Z80 PIO mentre RDY è basso, il dato precedente sarà sovrapposto dal nuovo e perduto — e non sarà rilevato nessun stato di errore.

TRASFERIMENTI DI DATI BIDIREZIONALI NELLO Z80 PIO CON "HANDSHAKING"

Nel modo bidirezionale (Modo 2), le linee di controllo di supporto alle Porte di I/O A e B sono entrambe applicate ai dati bidirezionali che sono trasferiti con la Porta A; la Porta B deve essere posizionata nel modo di controllo dei bits (Modo 3). La temporizzazione per trasferimenti di dati bidirezionali è semplicemente una combinazione di "handshaking" di ingresso e di uscita dove le linee di controllo A si applicano all'uscita dei dati mentre le linee di controllo B si applicano all'ingresso dei dati. Ciò può essere illustrato come segue:

La caratteristica unica della precedente illustrazione è che il dato che sta per uscire attraverso la Porta A è stabile solo per la durata dell'impulso basso di $\overline{A\ STB}$. Ciò è necessario nel modo bidirezionale poichè i pins della Porta A devono essere pronti a ricevere i dati in ingresso non appena l'operazione di uscita è completata.

Nuovamente, è compito della logica esterna assicurarsi che ciò sia conforme ai requisiti della temporizzazione del funzionamento in modo bidirezionale. La logica esterna deve leggere i dati in uscita mentre $\overline{A\ STB}$ è basso. Se la logica esterna non riporterà nessun stato di errore alla CPU; non c'è nessun segnale che la logica esterna rimanda allo Z80 PIO dopo una lettura positiva.

Inoltre è compito della logica esterna assicurarsi che essa trasmetta dati alla Porta A solo quando B RDY è alto e A RDY è basso. Se la logica esterna tenta di mettere in ingresso dei dati mentre lo Z80 PIO ne sta facendo uscire, i dati in ingresso non saranno accettati. Se la logica esterna tenta di mettere in ingresso dei dati prima che si siano letti i dati precedenti, il dato d'ingresso precedente sarà perduto e non sarà riportato nessun stato di errore.

CONTROLLO DEI BIT
I/O SEMPLICE

Il modo di Controllo (Modo 3) non usa segnali di controllo. Nel Modo 3 si deve definire ogni pin di una porta di I/O come pin d'ingresso e pin di uscita. L'ingresso e l'uscita sono controllati dalla CPU, non c'è nessun "handshaking" con

logica esterna. Se tutti i pin di una porta sono definiti nella stessa direzione, allora la porta può essere usata come semplice ingresso o uscita paralleli.

SELEZIONE DEL MODO DELLA PORTA DI I/O

I modi delle porte si scelgono scrivendo un codice appropriato nel buffer di controllo delle porte. Una discussione dettagliata dei codici di controllo non vi aiuterà a capire la materia soggetto di questo capitolo, così lasciamo quella discussione a An Introduction to Microcomputers: Volume II – Some Real Products.

INDIRIZZAMENTO DELLA PORTA DI I/O

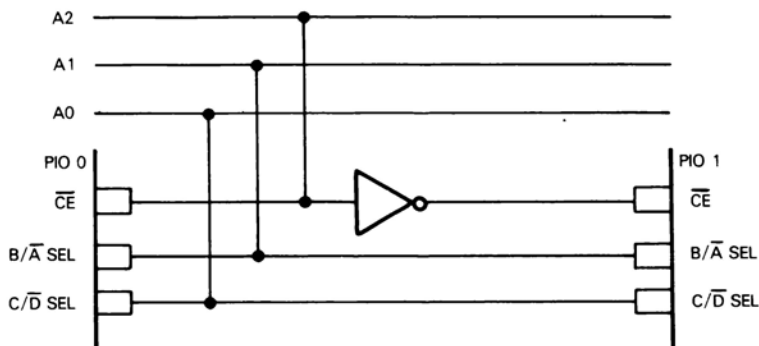
Ogni Z80 PIO ha quattro indirizzi delle porte di I/O assegnati ad esso. Tre pin dello Z80 PIO sono usati per scegliere il dispositivo ed una porta del dispositivo, come segue:

$\overline{CE}$: Ingresso 0 per scegliere il dispositivo. Ingresso 1 per scollegarlo.
B/$\overline{A}$ SEL: Ingresso 0 per scegliere la Porta A. Ingresso 1 per scegliere la Porta B.
C/$\overline{D}$ SEL: Ingresso 0 per scegliere il buffer del dato. Ingresso 1 per scegliere il buffer di controllo.

Ecco un sommario delle combinazioni delle selezioni del dispositivo:

SEGNALE			LOCAZIONE SELEZIONATA
$\overline{CE}$	B/$\overline{A}$ SEL	C/$\overline{D}$ SEL	
0	0	0	Buffer dati della porta A
0	0	1	Buffer di controllo della porta A
0	1	0	Buffer dati della porta B
0	1	1	Buffer di controllo della porta B
1	X	X	Dispositivo non selezionato

DETERMINAZIONE DELL'INDIRIZZO DELLA PORTA DI I/O

Ora quando si esegue un'istruzione IN o OUT da parte dello Z80 CPU, il numero della porta è fatto uscire sulle otto linee di ordine minore del Bus degli Indirizzi. Useremo due dispositivi Z80 PIO, e li collegheremo al Bus degli Indirizzi come segue:

In conseguenza dei collegamenti mostrati sopra, gli Z80 PIO risponderanno ai seguenti indirizzi delle porte di I/O:

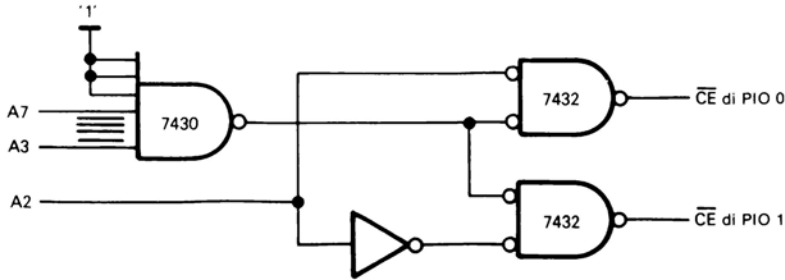
Per ragioni di coerenza, assegneremo sempre degli 0 ai bit "don't care". Le locazioni Z80 PIO saranno in tale modo indirizzate come segue:

INDIRIZZO	LOCAZIONE	MNEMONICO
0	Dati della porta A del PIO 0	A0
1	Controllo della porta A del PIO 0	AC0
2	Dati della porta B del PIO 0	B0
3	Controllo della porta B del PIO 0	BC0
4	Dati della porta A del PIO 1	A1
5	Controllo della porta A del PIO 1	AC1
6	Dati della porta B del PIO 1	B1
7	Controllo della porta B del PIO 1	BC1

Poichè i bit "don't care" potrebbero avere un valore qualunque, **abbiamo realmente usati tutti i 256 indirizzi delle porte di I/O per accedere a sole otto locazioni separate.** Poichè abbiamo bisogno di due soli Z80 PIO per il programma che svilupperemo, **questo schema di indirizzamento è soddisfacente per il nostro limitato scopo.**

Ci sono due modi per indirizzare più di una porta di I/O:

1) **Assegnare indirizzi di memoria ad ogni ulteriore porta di I/O,** come abbiamo mostrato nel Capitolo 2.
2) **Riservare proprio gli otto indirizzi richiesti per le porte di I/O per i due Z80 PIO.** Ciò significa che si deve aggiungere maggiore logica per decodificare un singolo segnale di abilitazione delle linee da A3 ad A7 del Bus degli Indirizzi.

   Ecco una logica per riservare gli indirizzi compresi tra $F8_{16}$ e FF_{16}:

Un 7430 è una porta NAND positiva a 8 ingressi; un 7432 è una porta OR positiva a due ingressi. Quando le cinque linee superiori d'indirizzo delle porte di I/O sono tutte 1, lo Z80 PIO selezionato da A2 riceverà uno 0 sul suo ingresso $\overline{CE}$.

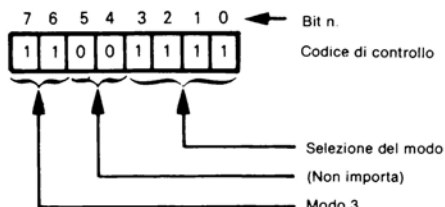
Inizialmente, per mantenere le cose semplici, programmeremo entrambi gli Z80 PIO per funzionare nel Modo 3, con la seguente assegnazione della direzione dei dati:

Z80 PIO	PORTA	PIN	DIREZIONE
0	A B	Tutti 7 - 4 3 - 0	Ingresso Ingresso Uscita
1	A B	Tutti Tutti	Uscita Ingresso

SEQUENZA DI ISTRUZIONI PER SELEZIONARE IL MODO DELLA PORTA DI I/O

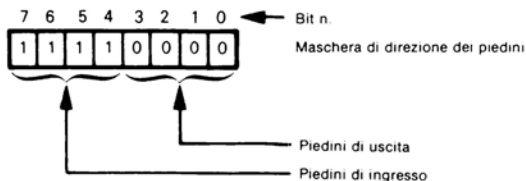
Per comprendere la discussione in corso, voi non dovete conoscere come lo Z80 PIO è stato programmato per andare incontro alle nostre esigenze; tuttavia, ecco un esempio dell'appropriata sequenza di istruzioni seguita da una spiegazione delle parole di controllo:

```
; Inizializzazione delle porte di I/O
        LD      B,0CFH    ; Mette la parola di controllo del Modo 3 nel regi-
                          ; stro B
; PIO 0, PORTA B
        LD      C,3       ; Mette l'indirizzo di controllo nel registro C
        OUT     (C),B     ; Posiziona la porta nel Modo 3
        LD      A,0F0H    ; Mette la parola della direzione dei pin nell'Accu-
                          ; mulatore
        OUT     (C),A     ; Posiziona le direzioni: ingressi la metà superiore,
                          ; uscite l'inferiore
; PIO 0, PORTA A
        –
        –
        –
```

La seguente parola di controllo fa sì che la porta indirizzata funzioni nel Modo 3:

Per verificare il formato della parola di controllo, vedere la descrizione dello Z80 PIO in An Introduction to Microcomputers: Volume II -- Some Real Products.

Abbiamo arbitrariamente scelto il Registro B per contenere questa parola di controllo, che sarà la stessa per tutte le porte di I/O; in tale modo noi carichiamo la parola di controllo nel Registro B una volta sola, all'inizio della sequenza che inizializza tutte le porte. Ciò è fatto con l'istruzione LD B,0CFH.

Carichiamo quindi l'indirizzo della porta di controllo nel Registro C con l'istruzione LD C,3 e quindi facciamo uscire la parola di controllo con l'istruzione OUT (C),B.

Se si fa uscire un codice di controllo "Mode Select" specificando che una porta d'I/O funzionerà nel Modo 3, allora si assume che il prossimo byte in uscita sia una maschera della direzione dei pin. Abbiamo usato la maschera seguente nell'esempio precedente:

Un 1 identifica un pin d'ingresso, uno 0 identifica un pin d'uscita. L'istruzione LD A,0F0H mette questa maschera di direzione dei pin nell'Accumulatore. L'istruzione OUT (C),A invia la maschera di direzione dei pin alla porta.

SEGNALI D'INGRESSO

Volgiamo la nostra attenzione ai segnali d'ingresso che compaiono nella parte sinistra della Figura 3-1. Descriveremo ogni segnale, lo assegneremo ad un appropriato pin d'ingresso, e includeremo una rudimentale sequenza d'istruzioni per accedere ai segnali a un livello molto elementare.

$\overline{\text{RETURN STROBE}}$

Se l'operatore deve vedere il carattere stampato più recentemente, devono accadere due cose:

1) La ruota di stampa deve essere spostata nella sua posizione di visibilità.

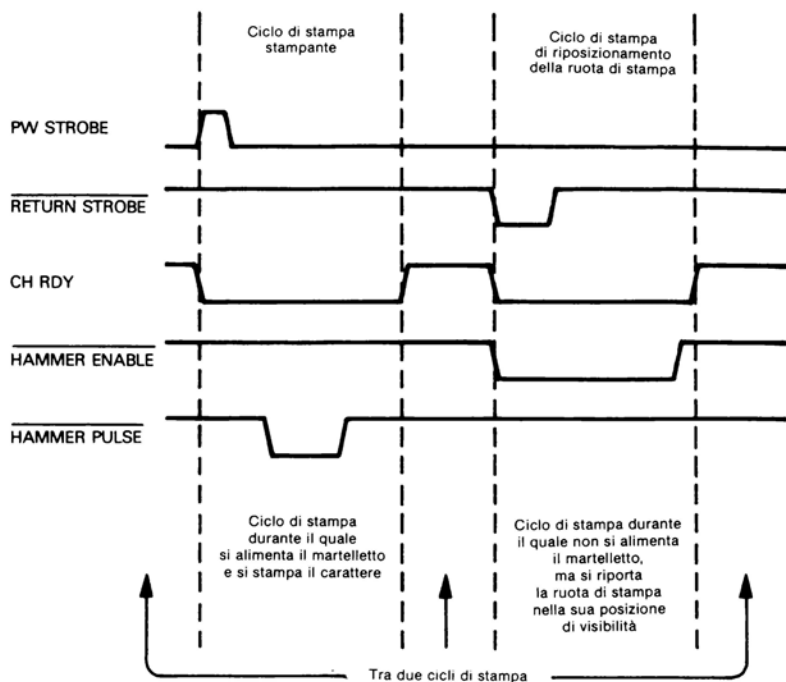
2) Il nastro deve essere fatto cadere.

La logica esterna può sorvegliare la caduta e la salita del nastro, ma la logica di Figura 3-1 crea i segnali che permettono alla ruota di stampa di muoversi.

Per muovere la ruota di stampa nella sua posizione di visibilità, la logica esterna di controllo del nastro mette l'ingresso $\overline{\text{RETURN STROBE}}$ basso mentre si fa cadere il nastro.

CICLO DI STAMPA DI RIPOSIZIONAMENTO DELLA RUOTA DI STAMPA

La logica della Figura 3-1 usa il $\overline{\text{RETURN STROBE}}$ come un segnale alternativo per dar inizio ad un ciclo di stampa; tuttavia il $\overline{\text{RETURN STROBE}}$ basso è accompagnato da $\overline{\text{HAMMER ENABLE FF}}$ basso, che impedice l'alimentazione del martelletto di stampa. Perciò, **un ciclo di stampa iniziato dal $\overline{\text{RETURN STROBE}}$ basso è un ciclo di stampa "simulato" che fa tornare la ruota di stampa, alla sua posizione di visibilità ma che non alimenta il martelletto di stampa; ci riferiremo a questo ciclo come al ciclo di stampa di riposizionamento della ruota di stampa.**

Assegneremo il pin 4 della porta di I/O B0 al segnale $\overline{\text{RETURN STROBE}}$.

Si può saggiare questo pin tra due cicli di stampa per dar inizio ad un nuovo ciclo di stampa mediante la seguente sequenza d'istruzioni:

```
LOOP:   IN    A,(2)        ; Il contenuto della porta di I/O B0 entra nell'Ac-
                           ; cumulatore
        BIT   4,A          ; Test sul valore del bit 4
        JR    NZ,LOOP      ; Se è 1, ritorna e ripeti il test
; La sequenza di istruzioni del nuovo ciclo di stampa comincia qui
```

PFL REL

Non si può alimentare il martelletto di stampa mentre il meccanismo che rifornisce la carta si sta muovendo, perciò in tali istanti la logica esterna mette basso l'ingresso PFL REL.

La logica nella Figura 3-1 ritarderà lo sparo del martelletto di stampa finchè l'ingresso PFL REL sarà basso.

Assegneremo a PFL REL il Pin 0 della Porta d'I/O A0.

Prima di eseguire la sequenza d'istruzioni che fa sparare il martelletto di stampa, metteremo in ingresso il contenuto della Porta A0 e saggeremo il bit 0; finchè il bit conterrà zero, non eseguiremo la sequenza che fa sparare il martelletto di stampa. Le istruzioni seguenti effettuano la prova richiesta:

```
LOOP:   IN    A,(0)        ; Fa entrare il contenuto della Porta di I/O A0 nel-
                           ; l'Accumulatore
        BIT   0,A          ; Saggia il valore del bit 0
```

```
         JR   Z,LOOP      ; Se il valore è 0, non alimentare il martelletto di
                          ; stampa
; La sequenza di istruzioni che alimenta il martelletto di stampa comincia qui
```

RIB LIFT RDY

Questo segnale è simile a PFL REL; è **basso in ingresso quando la logica di sollevamento del nastro sta muovendo il nastro.** Come il martelletto di stampa non può essere alimentato mentre è attivato il meccanismo di rifornimento della carta, così esso non può essere sparato mentre si sta muovendo il nastro. Collegando RIB LIFT RDY al Pin 1 della Porta di I/O A, possiamo adattare la sequenza di istruzioni per l'iniziazione dello sparo del martelletto di stampa come segue:

```
LOOP:    IN   A,(0)       ; Il contenuto della Porta di I/O A0 va nell'Accu-
                          ; mulatore
         OR   0FCH        ; Maschera tutti i bit eccetto il Bit 0 e il Bit 1
         CPL              ; Complementa il risultato per saggiare se è presente
                          ; qualche bit a 0
         JR   NZ,LOOP     ; Ogni bit 0 sarà ora a 1, se qualche bit è ora 1, non
                          ; sparare il martelletto di stampa
; La sequenza di istruzioni per far alimentare il martelletto di stampa comincia qui
```

PW STROBE

Abbiamo già incontrato **questo segnale; è messo alto impulsivamente da una logica esterna per dar inizio ad un normale ciclo di stampa,** durante il quale si stampa un carattere.

Ricorderete che $\overline{\text{RETURN STROBE}}$ è messo basso in ingresso per inizializzare un ciclo di stampa, durante il quale la ruota di stampa sarà spostata nella sua posizione di visibilità, ma nel quale non si stamperà nessun carattere.

Supponendo che **PW STROBE sia collegato al pin 5 della Porta di I/O B0,** ecco la sequenza di istruzioni che verrà eseguita tra due cicli di stampa:

```
LOOP:    IN   A,(2)       ; Il contenuto della Porta di I/O di ingresso B0 va
                          ; nell'Accumulatore
         AND  30H         ; Isola i bit 5 (PW STROBE) e 4 (RETURN STROBE)
         CP   10H         ; Test per PW STROBE = 0, RETURN STROBE = 1
         JR   Z,LOOP      ; Se il test è vero si rimane in LOOP
; La sequenza di istruzioni del ciclo di stampa comincia qui
```

Osservare che sia PW STROBE = 1 che $\overline{\text{RETURN STROBE}}$ = 0 possono dare l'avvio ad un ciclo di stampa; ecco perchè solo PW STROBE = 0 e $\overline{\text{RETURN STROBE}}$ = 1 ci mantengono nel LOOP delle istruzioni che fanno il test.

DURATA DELL'IMPULSO DEL SEGNALE D'INGRESSO

Ora le istruzioni mostrate sopra vengono eseguite in un numero complessivo di 36 cicli di clock. Con un clock di 500ns, le quattro istruzioni saranno eseguite in 18 microsecondi — che diventa la minima durata d'impulso consentita per PW STROBE. **Se PW STROBE ha un impulso alto per meno di 18 microsecondi, il nostro ciclo di istruzioni può perderlo.**

$\overline{\text{FFA}}$

Questo è un altro **segnale** di avviso per il martelletto di stampa. **E' posto a zero mentre la logica esterna sta facendo avanzare il nastro. Collegando il segnale al pin 2 della Porta di I/O A0, possiamo modificare la sequenza di istruzioni che precede**

l'alimentazione del martelletto di stampa come segue:

```
LOOP:   IN    A,(0)        ; Il contenuto della Porta di I/O A0 entra nell'accu-
                           ; mulatore
        OR    0F8H         ; Isola i bit 2, 1 e 0
        CPL                ; Complementa il risultato per saggiare se qualche
                           ; bit è a 0
        JR    NZ,LOOP      ; Ora ogni bit A0 sarà 1. Se qualche bit è 1, non
                           ; mettere in fila il martelletto di stampa
, La sequenza che fa sparare il martelletto di stampa comincia qui
```

Tutto ciò che abbiamo fatto è di avere aggiunto una condizione da verificare, che deve essere verificata prima di eseguire la sequenza d'istruzioni che fa sparare il martelletto di stampa.

RESET

Questo segnale si vede comunemente in diversi tipi di logica. E' **un segnale di inizializzazione.** Il süo scopo è di assicurare che tutta la logica si trovi in uno stato "iniziale", che nel nostro caso è la condizione che esiste tra due cicli della ruota di stampa.

La logica della Figura 3–1 connette il segnale di RESET ai dispositivi logici, in modo che quando il segnale RESET va alto forza tutta la logica ad una condizione "iniziale".

RESET DELLA CPU	**Ci sono molti modi con i quali un sistema a microcalcolatore può maneggiare un segnale di RESET. Lo schema più semplice è di mettere questo segnale in ingresso al pin $\overline{\text{RESET}}$ della Z80 CPU.**

Un altro metodo di maneggiare il RESET è di saggiare il segnale tra due cicli di stampa e di impedire l'inizio di un ciclo di stampa mentre il RESET è alto; ciò può essere fatto collegando il RESET al pin 6 della Porta di I/O B0 e modificando quindi la nostra sequenza d'istruzioni "tra due cicli di stampa" come segue:

```
LOOP:   IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
        BIT   6,A          ; test sul bit 6 (RESET)
        JR    NZ,LOOP      ; Se il RESET è alto, si rimane in LOOP
; Il RESET è basso. Test su PW STROBE e RETURN STROBE
        AND   30H          ; Isola i bit 5 (PW STROBE) e 4 (RETURN STROBE)
        CP    10H          ; Test per vedere se PW STROBE = 0, RETURN
                           ; STROBE = 1
        JR    Z,LOOP       ; Se il test è vero si rimane in LOOP
; La sequenza d'istruzioni del ciclo di stampa comincia qui
```

DURATA DELL'IMPULSO DEL SEGNALE	**Questo loop di test più lungo richiede ora 51 cicli per essere eseguito. Ciò significa che PW STROBE deve avere un impulso alto per almeno 25,5 microsecondi, supponendo un clock di 500 nanosecondi.**

PFR REL

Questo è un altro **segnale** che deve essere saggiato prima di alimentare il martelletto. Esso **indica che la logica esterna sta facendo muovere il rifornimento di carta.** In queste circostanze, non possiamo alimentare il martelletto di stampa. **Collegando questo segnale al pin 3 della Porta di Ingresso A0, dobbiamo semplicemente adattare la sequenza d'istruzioni che fa sparare il martelletto di stampa come segue:**

```
LOOP:   IN    A,(0)       ; Il contenuto della porta di I/O entra nell'accu-
                          ; mulatore
        OR    0F0H        ; Isola i bit 3, 2, 1 e 0
        CPL               ; Complementa il risultato per saggiare se qualche
                          ; bit è a 0
        JR    NZ,LOOP     ; Bit a 0 sarà ora 1. Se qualche bit è 1, non alimen-
                          ; tare il martelletto di stampa
; La sequenza che fa sparare il martelletto di stampa comincia qui
```

CA REL

Questo segnale è quasi identico a PFR REL. Esso **proviene dalla logica esterna che controlla il movimento del carrello. Collegheremo questo segnale al pin 4 della Porta di Ingresso A0** e modificheremo la sequenza d'inizializzazione delle istruzioni che alimentano il martelletto come segue:

```
LOOP:   IN    A,(0)       ; Il contenuto della Porta di I/O A0 entra nell'ac-
                          ; cumulatore
        OR    0E0H        ; Isola i bit 4, 3, 2, 1 e 0
        CPL               ; Complementa il risultato per provare se qualche
                          ; bit è 0
        JR    NZ,LOOP     ; Ogni bit a 0 sarà ora 1. Se qualche bit è 1 non
                          ; alimentare il martelletto di stampa
; La sequenza che alimenta il martelletto di stampa comincia qui
```

FFI

Questo è un segnale che temporizza il primo ritardo nel ciclo di stampa — il tempo durante il quale la ruota di stampa si muove dalla sua posizione di visibilità finchè il petalo richiesto non sia di fronte al martelletto di stampa.

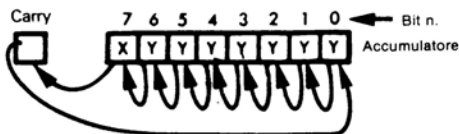
FFI è generato dalla logica esterna; è basso mentre la ruota di stampa si sta muovendo e alto mentre la ruota di stampa non si sta muovendo.

RITARDO DI TEMPO BASATO SUL SEGNALE D'INGRESSO

Collegheremo FFI al pin 7 della Porta di I/O A0. Il seguente loop di istruzioni creerà un ritardo che dura finchè FFI non va alto:

```
LOOP:   IN    A,(0)       ; La porta A0 entra nell'accumulatore
        RLA               ; Sposta il bit 7 del riporto (Carry)
        JR    NC,LOOP     ; Se Carry = 0, rimanere in LOOP
```

Avete visto come funziona questo loop? Dopo che il contenuto della Porta di I/O A0 è entrato nell'Accumulatore, ora ci interessa solo il bit 7, poichè esso è il bit che corrisponde a FFI.

Ecco cosa fa l'istruzione RLA:

Se lo stato del Carry è 1, il ritardo del movimento della ruota di stampa è finito. Se il Carry è 0, la logica del programma deve far continuare il ritardo.

Perchè si usa un'istruzione RLA per saggiare questo bit invece che un'istruzione BIT? L'istruzione BIT usa due byte di codice oggetto ed un tempo di esecuzione di otto cicli di clock, mentre l'istruzione RLA un solo byte ed è eseguita in quattro cicli di clock.

$\overline{\text{EOR DET}}$

Questo segnale indica che si è raggiunto la fine del nastro. In queste circostanze la stampa dei caratteri non può continuare.

Quando si genera questo segnale, c'è ancora del nastro nuovo di fronte al martelletto di stampa, così che il segnale non è usato per impedire l'alimentazione del martelletto di stampa; piuttosto, esso è usato per impedire che non venga mai indicata la fine del ciclo di stampa. Ciò in effetti impedisce che venga iniziato un nuovo ciclo di stampa.

Collegheremo il segnale $\overline{\text{EOR DET}}$ al bit 7 della Porta di I/O B0. Poichè $\overline{\text{EOR DET}}$ è un segnale logico negato, lo saggeremo prima di entrare nel loop "tra due cicli di stampa" come segue:

```
; Test di fine valida di un ciclo di stampa
VALND: IN    A,(2)        ; La Porta di I/O B0 entra nell'accumulatore
       RLA                ; Sposta il bit 7 nel Carry
       JR    NC,VALND     ; Se nel Carry c'è zero, rimanere nel ciclo di stampa
; Inizio del loop "tra due cicli di stampa"
LOOP:  IN    A,(2)        ; La Porta di I/O B0 entra nell'accumulatore
       BIT   6,A          ; Test sul bit 6 (RESET)
       JR    NZ,LOOP      ; Se il RESET è alto, rimani in loop
; Il RESET è basso. Saggiare PW STROBE e RETURN STROBE
       AND   30H          ; Isola i bit 5 (PW STROBE) e 4 (RETURN STROBE)
       CP    10H          ; Test per PW STROBE = 0, RETURN STROBE = 1
       JR    Z,LOOP       ; Se il test è vero rimanere in loop
; La sequenza delle istruzioni dei cicli di stampa comincia qui
```

Guardate la sequenza d'istruzione precedente. In essa ci sono alcuni aspetti interessanti.

Le prime tre precedenti istruzioni saranno le ultime tre istruzioni nella sequenza del ciclo di stampa. L'istruzione avente LOOP come label è la prima istruzione di una sequenza che viene eseguita continuamente finchè non inizia il prossimo ciclo di stampa. In tale modo, se $\overline{\text{EOR DET}}$ è basso, la logica del programma dipenderà dalle tre istruzioni elencate sopra, rimanendo in loop in queste tre istruzioni finchè $\overline{\text{EOR DET}}$ non va alto. In questo istante finisce il ciclo di stampa ed entriamo nel loop d'istruzioni "tra due cicli di stampa". Ora il programma sarà sospeso indefinitamente in questo loop di istruzioni finchè il bit 6 (che corrisponde a RESET) non sia 0, mentre il bit 5 (che corrisponde a PW STROBE) è 1, o il bit 4 (che corrisponde a RETURN STROBE) è 0.

C'è un altro aspetto interessante nella sequenza d'istruzioni precedente. Noi potremmo, se fosse desiderato, eliminare la seconda istruzione IN, come segue:

```
; Test di fine valida di un ciclo di stampa
VALND: IN    A,(2)        ; La Porta di I/O B0 entra nell'accumulatore
       RLA                ; Sposta il bit 7 nel Carry
       JR    NC,VALND     ; Se nel Carry c'è zero, rimanere nel ciclo di stampa
; Inizio del LOOP "trá due cicli di stampa"
       BIT   7,A          ; Test sul bit 6 (RESET)
       JR    NZ,VALND     ; Se RESET è alto, rimanere in LOOP
; RESET è basso, saggiare PW STROBE e RETURN STROBE
```

```
         AND  60H          ; Isola i bit 5 (PW STROBE) e 4 (RETURN STROBE)
         CP   20H          ; Test per PW STROBE = 0, RETURN STROBE = 1
         JR   Z,VALND      ; Se il test è vero rimanere in LOOP
; La sequenza delle istruzioni di cicli di stampa comincia qui
```

Eliminando una sola istruzione, abbiamo risparmiato due byte di codice oggetto. La penalizzazione è di aver aggiunto 11 cicli di clock all'intero loop di istruzioni, il che significa che l'impulso alto di PW STROBE supera i 25,5 microsecondi calcolati nella discussione del segnale di RESET arrivando a 31 microsecondi.

Perchè si fa lavorare la sequenza di istruzioni condensata illustrata sopra? La ragione è che si è supposto che la logica esterna non faccia muovere il nastro tra due cicli di stampa; perciò EOR DET sarà sempre alto durante il loop d'esecuzione delle istruzioni "tra due cicli di stampa". Se fosse così, l'istruzione RLA sposterà sempre un 1 nel Carry, che farà sempre in modo che l'esecuzione continui con l'istruzione BIT. In tale modo, le prime tre istruzioni diventano innocue. Notare che gli operandi delle istruzioni BIT, AND e CP sono cambiati, poichè tutti i bit sono stati spostati di una sola posizione a sinistra dell'istruzione RLA.

HAMMER ENABLE FF

Questo è il segnale che impedisce al martelletto di stampa di essere alimentato dopo che la ruota di stampa si è mossa nella sua posizione di visibilità, come descritto congiuntamente al segnale RETURN STROBE.

Collegheremo HAMMER ENABLE FF al pin 6 della Porta di I/O A0, quindi modificheremo la sequenza d'istruzioni che precede l'alimentazione del martelletto di stampa come segue:

```
LOOP:    IN   A,(0)        ; Il contenuto della Porta di I/O A0 entra nell'ac-
                           ; cumulatore
         OR   0A0H         ; Isola i bit 6, 4, 3, 2, 1 e 0
         CPL               ; Complementa il risultato per saggiare se qualche
                           ; bit è 0
         JR   NZ,LOOP      ; Ogni bit a 0 ora sarà 1. Se qualche bit è 1, non
                           ; alimentare il martelletto di stampa
; La sequenza che fa sparare il martelletto di stampa comincia qui
```

CLK

Questo è il segnale di clock che sincronizza tutta la logica della Figura 3-1. Non possiamo includere questo segnale nella nostra simulazione di Figura 3-1, nonostante tutti i possibili tentativi, poichè gli eventi nel programma del microcalcolatore saranno sincronizzati dalla sequenza con la quale vengono eseguite le istruzioni — non da un clock. **Analogamente i due prossimi segnale +5V e RV1, sono alimentazioni. Essi sono senza significato in un programma di microcalcolatore.**

H1 — H6

Questi sono i sei segnali che scelgono sei durate di tempo per l'impulso che alimenta il martelletto di stampa. Assegneremo questi segnali alla Porta di I/O B1. Una volta che è stata eseguita la sequenza d'istruzioni che alimenta il martelletto di stampa, essa carica semplicemente questi segnali nell'Accumulatore come segue:

```
         IN   A,(6)        ; Il codice del tempo dell'impulso per l'alimentazio-
                           ; ne entra nell'Accumulatore
```

SOMMARIO DEI SEGNALI DI INGRESSO

In sommario ecco come sono stati assegnati i segnali d'ingresso:

La porta A dello Z80 PIO 0 (Porta A0) è assegnata in ingresso

Bit	Segnale
7	FFI
6	$\overline{\text{HAMMER ENABLE}}$
5	
4	CA REL
3	PFR REL
2	$\overline{\text{FFA}}$
1	RIB LIFT RDY
0	PFL REL

La porta B dello Z80 PIO 0 (Porta B0) è assegnata in ingresso

Bit	Segnale
7	$\overline{\text{EOR DET}}$
6	RESET
5	PW STROBE
4	$\overline{\text{RETURN STROBE}}$

La porta B dello Z80 PIO 1 (Porta B1) è assegnata in ingresso

Bit	Segnale
7	
6	
5	H6
4	H5
3	H4
2	H3
1	H2
0	H1

SEGNALI DI USCITA

Volgeremo ora la nostra attenzione ai segnali di uscita elencati sulla destra della Figura 3–1. Questi segnali sono molto più facili da descrivere che i segnali d'ingresso. Essi consistono in sei uscite di flip-flop – che sono semplicemente degli indicatori di temporizzazioni usati dalla logica esterna – più quattro segnali di controllo. **Faremo uscire questi segnali sulla Porta B di uno Z80 PIO e sulla Porta A di un secondo Z80 PIO, come segue:**

Assegnamo un pin a FFC anche se esso non è un'uscita, perchè la Porta di I/O A1 servirà a un duplice scopo — come locazione di memorizzazione di un dato e come buffer di segnali di uscita. Non si possono mescolare semplici programmi per generare segnali di uscita; ciò è lo scopo della logica di Figura 3-1. Perciò definiremo semplicemente i quattro segnali di controllo di uscita:

1) **PW REL. Questo segnale segna la fine del ritardo di tempo di assestamento e di ritorno del martelletto di stampa e l'inizio del ritardo fisso del Movimento Finale** durante il quale la logica esterna può far muovere il rifornimento della carta e il carrello.

2) **CH RDY. Questo è pure riportato come il segnale PRINTWHEEL READY. E' il segnale che definisce l'intero intervallo di tempo di ciclo di stampa;** esso va basso all'inizio del ciclo di stampa e rimane basso fino alla fine del ciclo di stampa.

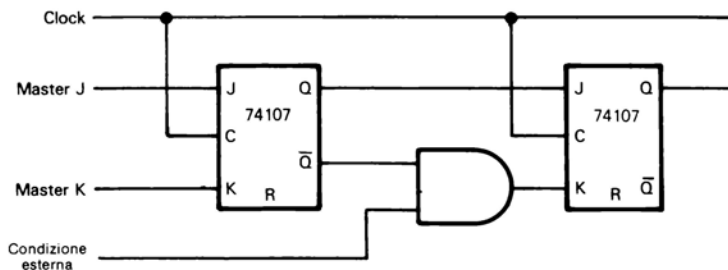
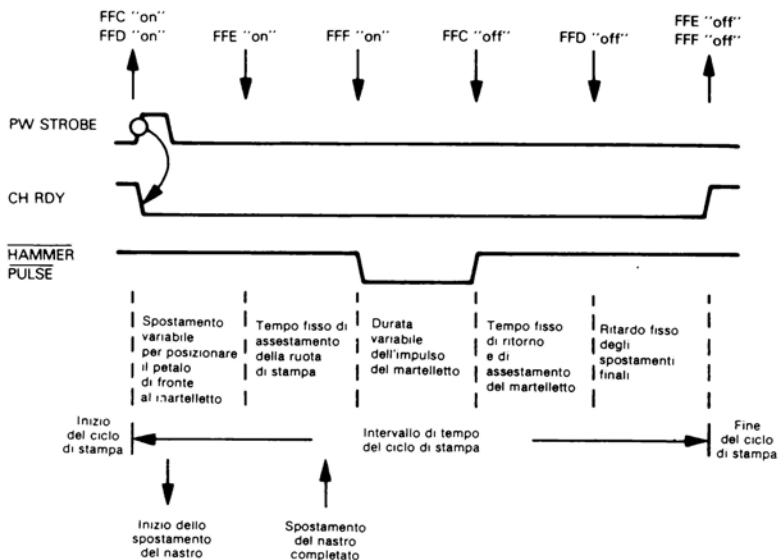
3) **$\overline{\text{HAMMER PULSE}}$. Questo segnale è fatto uscire basso per l'intervallo di tempo durante il quale la logica esterna trasmette un impulso di alimentazione al solenoide del martelletto di stampa.**

4) **START RIBBON MOTION PULSE. Questo segnale ha un impulso alto all'inizio del ciclo di stampa, per dire alla logica esterna che è libero di cominciare a fare avanzare il nastro così che quando si alimenterà il martelletto di stampa ci sia nastro nuovo di fronte ad esso.**

SIMULAZIONE ORIENTATA VERSO LA LOGICA DIGITALE

Siamo ora pronti a simulare la logica illustrata in Figura 3-1 — ma prima diamo una breve occhiata alla logica.

UNA PANORAMICA SULLA LOGICA

Al centro della sequenza logica ci sono quattro flip-flop 74107, etichettati FFC_W, FFD_W, FFE_W e FFF_W. Troverete questi flip-flop nel centro e sulla sinistra della Figura 3-1. **Questi quattro flip-flop formano il contatore noto come "Contatore di Johnson".** Ogni flip-flop è controllato dall'uscita del flip-flop precedente, accoppiato ad un test su condizioni esterne:

In tale modo i quattro flip-flop possono essere visualizzati come eventi che danno inizio a cicli di stampa nel modo seguente:

Come illustrato sopra, l'intervallo di tempo di un ciclo di stampa, può essere diviso in cinque periodi.

Durante il primo intervallo di tempo, la ruota di stampa è fatta muovere dalla sua posizione di visibilità fino a che il petalo voluto non sia di fronte al martelletto di stampa. Questo intervallo di tempo è controllato dalla logica esterna mediante l'ingresso FFI.

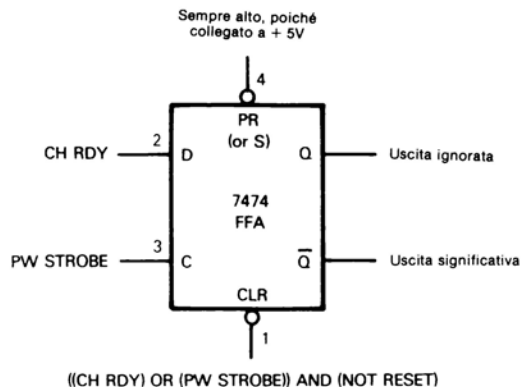
I rimanenti quattro intervalli di tempo sono controllati da tre monostabili 74121 e da un multivibratore 555.

Che cosa si può dire sui due flip-flop 7474 in alto a sinistra della Figura 3-1? Essi sono semplice logica d'inizializzazione del ciclo. Il flip-flop FFA è innescato da una combinazione di segnali necessari per l'inizio di un ciclo di stampa. Il flip-flop FFB agisce come un interruttore per i quattro flip-flop 74107, forzandoli a "chiudersi" tra due cicli di stampa. Il flip-flop FFB fa ciò collegando la sua uscita Q agli ingressi di reset dei flip-flop 74107. L'effetto di ciò è che il flip-flop 74107 sono sempre "Off" se FFB è "Off"; più tardi spiegheremo in maggior dettaglio come ciò accada.

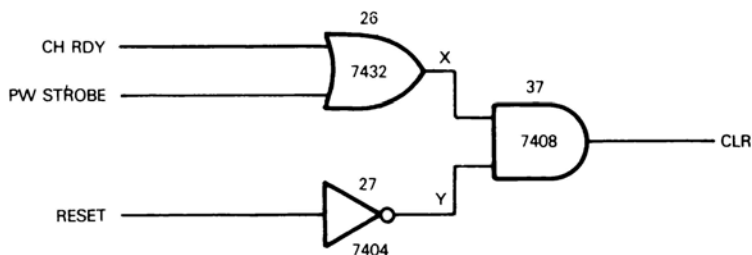
Seguiremo ora un ciclo di stampa nella Figura 3-1. Mano a mano che avanzeremo, creeremo un programma sul microcalcolatore in linguaggio assembly che simulerà la logica, dispositivo per dispositivo.

FLIP-FLOP FFA_W

FLIP-FLOP 7474

Il nostro ciclo di stampa comincia col flip flop 7474 indicato come FFA_W. Troverete questo flip-flop in alto a sinistra della Figura 3-1. Isoliamo FFA_W e illustriamolo come segue:

Torniamo a riferirci alla tabella generale di funzionamento di un flip-flop 7474 data nel Capitolo 2.

Poichè il PRESET (PR) è sempre alto, essendo collegato a +5V, un impulso basso sull'ingresso CLEAR (CLR) forzerà il flip-flop nello stato di "off", dopo di che l'uscita Q è bassa e l'uscita $\overline{Q}$ è alta.

Guardate la Figura 3-1 e vedrete che CLR è generato come segue:

Questa è la tabella della verità di CLR:

CH RDY	PW STROBE	X	RESET	Y	CLR
0	0	0	0	1	0
			1	0	0
0	1	1	0	1	1
			1	0	0
1	0	1	0	1	1
			1	0	0
1	1	1	0	1	1
			1	0	0

Per mettere "on" il flip-flop FFA_W, CLR deve essere alto; perchè CLR sia alto, il RESET deve essere basso e l'uno o l'altro di CH RDY o PW STROBE deve essere alto.

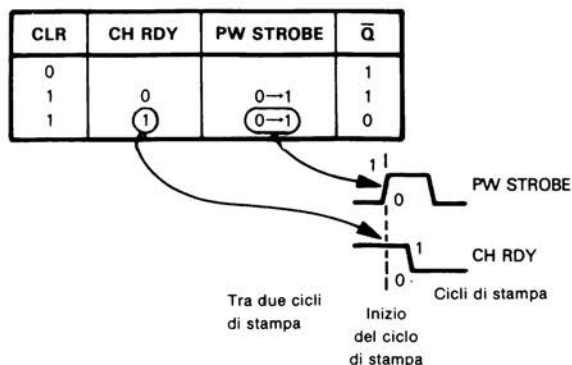
Ora CH RDY fornisce il dato all'ingresso (D) di FFA_W e PW STROBE fornisce lo ingresso di clock (C). Perciò la tavola di funzionamento del flip-flop FFA_W può essere illustrata come segue:

INGRESSI				USCITE		
PRESET	CLR	CLOCK (PW STROBE)	D (CH RDY)	Q	$\overline{Q}$	
0	1	0 o 1	0 o 1	1	0	PRESET=1
1	0	0 o 1	0 o 1	0	1	
0	0	0 o 1	0 o 1	Instabile		PRESET=1
1	1	0→1	1	1	0	
1	1	0→1	0	0	1	
1	1	0	0 o 1	Precedente Q	Precedente $\overline{Q}$	Nessun cambiamento

E questa si riduce alla seguente piccola tabella di funzionamento:

CLR	CH RDY	PW STROBE	$\overline{Q}$	
0			1	condizione "off"
1	0	0→1	1	possibili condizioni "on"
1	1	0→1	0	possibili condizioni "on"

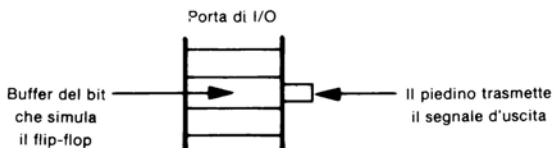
Ci vuole una transizione zero a uno di PW STROBE per mettere "on" il flip-flop FFA_W. Quando FFA_W va "on", tuttavia, se CH RDY è 0 allora l'uscita $\overline{Q}$ sarà ancora 1, rappresentando la condizione "off" quindi per mettere "on" FFA_W, PW STROBE deve andare da 0 a 1 mentre CH RDY è 1.

Ricordiamo che CH RDY è un segnale che è fatto uscire alto tra due cicli di stampa e che è fatto uscire basso per la durata di un ciclo di stampa. Ciò significa che il flip-flop FFA_W andrà "on" solo se PW STROBE ha un impulso alto tra due cicli di stampa, quando l'uscita CH RDY è alta:

Per il momento non preoccupiamoci su come CH RDY va subito a 0 dopo che il flip-flop FFA_W è andato "on"; spiegheremo come ciò accada più tardi. L'unica cosa importante da sottolineare è che un impulso alto di PW STROBE sarà ignorato se esso avviene mentre CH RDY è basso.

RESET **Che cosa si può dire sul segnale di RESET? Esso controlla tutta l'altra logica associata al flip-flop FFA_W; ogni volta che RESET è posto alto, si forza CLR basso che porta "off" il flip-flop FFA_W indipendentemente da ciò che esso sta facendo.**

SIMULAZIONE DEL FLIP-FLOP FFA_W

Abbiamo concluso nel Capitolo 2 che un flip-flop è rappresentato in un sistema con microcalcolatore da un singolo bit di una memoria a lettura e scrittura. Un singolo bit di un buffer a lettura e scrittura andrà pure bene.

SIMULAZIONE DEL FLIP-FLOP USANDO PORTE DI I/O **La Porta di I/O A1 è stata assegnata a segnali d'uscita.** Questa porta ha un buffer di 8 bit ai quali sono collegati i pin della porta; così **ogni bit del buffer della porta simulerà il flip-flop la cui uscita è trasmessa attraverso il pin della porta:**

Ricordiamo che ad FFA è stato assegnato il pin 0 della Porta di I/O A1.
Siamo pronti per simulare il flip-flop FFA_W.

Al tempo stesso, che cosa si può dire sulla simulazione delle tre porte sotto e sulla sinistra di FFA_W? Queste tre porte sono numerate 26, 27 e 37 ed insieme esse creano l'ingresso CLR.

Per simulare queste tre porte individualmente, si applica la seguente sequenza di istruzioni:

```
; Simulazione della Porta 27
        IN    A,(2)       ; Il contenuto della Porta di I/O B0 entra nel Reg. A
        CPL               ; Complementa tutti gli otto bit
        LD    B,A         ; Salva il complemento nel Registro B
; Simulazione della Porta 26
        CPL               ; Ricomplementa (rimemorizza) il contenuto del
                          ; Registro A
        AND   22H         ; Isola i bit 5 e 1; essi rappresentano PW STROBE
                          ; e CH RDY
; Simulazione della Porta 37
        JR    Z,CLR0      ; Se né il bit 1 né il bit 5 = 1, CLR è 0
        BIT   6,B         ; Saggia il complemento di RESET
        JR    Z,CLR0      ; Se il risultato è 0, CLR è 0
        SCF               ; CLR è 1 così si memorizza 1 nello stato di Carry
        JR    FFAW+2
CLR0:   AND   A           ; CLR è 0 così si memorizza 0 nello stato di Carry
; Simulazione del flip-flop FFAW
FFAW:   JR    NC,FFA0     ; Se CLR = 0, posiziona nella Porta A1, il bit 0 a 1
        BIT   5,A         ; CLR non è 0. Test su PW STROBE. Se PW STROBE
                          ; è 0, non si dà impulso di clock
```

```
        JR    Z,FFA0     ; Posiziona a 1 il bit 0 della Porta di I/O A1
        BIT   1,A        ; PW STROBE è ad 1. Verifica CH RDY
        JR    Z,FFA0     ; Se CH RDY = 0, poni ad 1 il bit 0 della Porta
                         ; I/O A1
        IN    A,(4)      ; Carica la Porta di I/O A1 nel Reg. A
        RES   0,A        ; Il bit 0 deve essere posizionato a zero; poichè
                         ; FFA è "on"
        OUT   (4),A
        JR    FFB        ; Salta alla simulazione del flip-flop B
FFA0:   IN    A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        SET   0,A        ; Il bit 0 deve essere posizionato a 1 poichè FFA
                         ; è "off"
        OUT   (4),A
; Segue la simulazione del flip-flop FFB
```

E' molto importante capire come le istruzioni si adattino insieme per fare un programma. Non dovete leggere oltre finchè non avrete compreso completamente come la sequenza d'istruzioni data sopra simuli la logica di FFA_W e delle sue tre porte associate.

Diamo un'occhiama alle simulazioni precedenti.

SIMULAZIONE DELL'INVERTITORE

Il segnale RESET, lo ricordate, è stato collegato al bit 6 della Porta di I/O B0 di uno Z80 PIO; questa porta è indirizzata come Porta 2 basandoci sul modo con cui abbiamo scelto di collegare lo Z80 PIO nel nostro sistema con microcalcolatore. Per invertire questo segnale, facciamo entrare il contenuto della Porta di I/O B0 nell'Accumulatore e complementiamo il contenuto dell'Accumulatore:

Il complemento di RESET e di tutti gli altri bit della Porta B0, è salvato nell'Accumulatore B. **Si è completata la simulazione della porta 27.**

SIMULAZIONE DELLA PORTA OR

FLAG DI STATO USATI PER RAPPRESENTARE DELLA LOGICA

La simulazione della porta 26 non è così immediata. Stiamo cercando l'OR di PW STROBE e CH RDY. Questi due segnali sono rappresentati dai bit 5 e 1, rispettivamente, della Porta B di I/O. Ora ciò che faremo è di memorizzare il contenuto della Porta di I/O B0 nell'Accumulatore complementando di nuovo il suo contenuto:

	$\overline{XXXXXXXX}$	Contenuto dell'Accumulatore
CPL	XXXXXXXX	Complemento

Quindi eseguiamo un'istruzione AND che posiziona a zero tutti i bit, tranne i bit 5 e 1. Ma noi dobbiamo in realtà fare l'OR di questi due bit rimanenti. Perchè? La ragione è che quando si esegue l'istruzione AND, essa posiziona lo stato Zero del complemento di (PW STROBE) in OR con (CH RDY).

A5 OR A1	A7	A6	A5	A4	A3	A2	A1	A0	VAL. ESAD.	STATO ZERO
	Contenuto dell'Accumulatore A									
0	0	0	0	0	0	0	0	0	00	1
1	0	0	0	0	0	0	1	0	02	0
1	0	0	1	0	0	0	0	0	20	0
1	0	0	1	0	0	0	1	0	22	0

PW STROBE (A5) — CH RDY (A1) — Dopo l'esecuzione dell'istruzione AND, lo stato Zero è il completamento di (PW STROBE) o (CH RDY)

STATO ZERO

Possiamo perciò spostarci sulla porta 37. Lo scopo della porta 37 è di generare l'ingresso CLR di FFA_W. Simuleremo CLR usando lo stato di Carry. Ora siamo appena venuti nella simulazione della porta 37 dalla simulazione della porta 26; a questo punto lo stato Zero sarà 0 se l'OR di PW STROBE con CH RDY è 1; altrimenti lo stato Zero sarà 1. (Ricordate che gli stati Zero rappresentano sempre l'inverso della condizione 0. In altre parole, una condizione 0 fa sì che lo stato Zero sia posizionato a 1; una condizione diversa da zero fa sì che lo stato Zero sia posizionato a 0).

La prima istruzione della simulazione della porta 37 si avvantaggia per il fatto che abbiamo l'OR di PW STROBE con CH RDY memorizzato nello stato Zero. Se lo stato Zero è 1, CLR deve essere 0, così la prima istruzione JR Z salta ad una logica che posizionerà lo stato di Carry a 0. L'istruzione successiva nella simulazione della porta 37 saggia il complemento di RESET memorizzato nel Registro B. Usando una istruzione BIT. L'istruzione BIT non cambierà il contenuto del Registro B, ma essa posizionerà lo stato Zero per riflettere il contenuto del bit 6. Se il complemento del RESET è 0, allora l'istruzione JR Z che segue salterà ad una logica di programma che posiziona lo stato di Carry a 0. Se il complemento di RESET è diverso da 0, allora si sono incontrate condizioni sulla porta 37 per far uscire un risultato diverso da zero — e questa condizione è simulata dall'istruzione SCF, che posiziona lo stato di Carry a 1.

Successivamente si simula il flip-flop FFA. Lo stato di questo flip-flop può essere definito come segue:

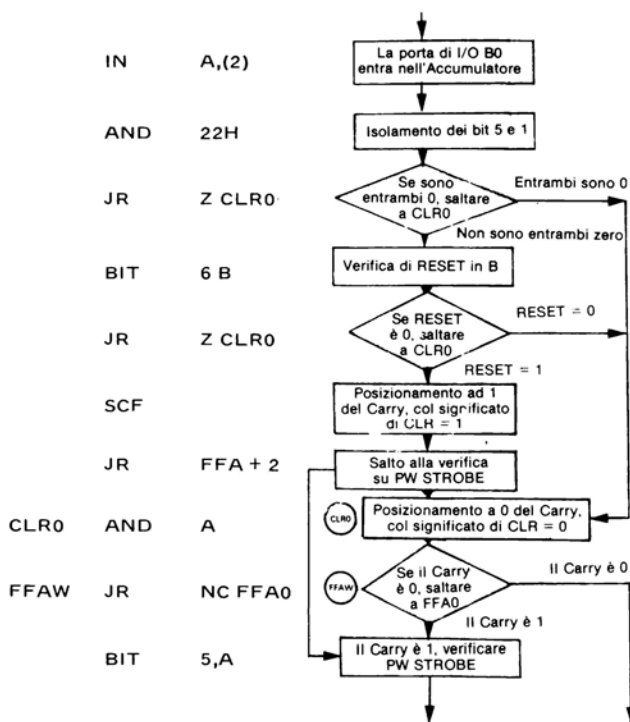
> Se CLR è 0 allora $\overline{Q}$ è 1.
> Se PW STROBE è 0 allora $\overline{Q}$ è 1.
> Se CLR è 1 e PW STROBE è 1 e CH RDY è 0 allora $\overline{Q}$ é 1.
> Se CLR è 1 e PW STROBE è 1 e CH RDY è 1 allora $\overline{Q}$ è 0.

CLR è simulato dallo stato di Carry. PW STROBE è simulato dal bit 5 dell'Accumulatore. CH RDY è simulato dal bit 1 dell'Accumulatore.

La simulazione del flip-flop FFA comincia con l'istruzione etichettata FFA_W.

STATO DI CARRY

Dapprima saggiamo lo stato di CLR usando l'istruzione JR NC. Questa istruzione provoca un salto a FFA0 se lo stato di Carry è 0 — che significa che CLR è 0. FFA0 è la label della prima istruzione nella sequenza che posiziona $\overline{Q}$ a 1.

Osservate che a questo punto nel programma abbiamo alcuni passi non necessari. Ecco la nostra logica:

Ogni rettangolo rappresenta una operazione di manipolazione o di movimento di dati.

Ogni rombo rappresenta una logica che saggia la condizione del flag di stato.

La sequenza logica illustrata sopra mantiene un flusso ordinato di istruzioni che si adatta al flip-flop FFA_W e alle sue tre precedenti porte. Ma se guardate le istruzioni etichettate CLR0 e FFAW, vedrete che sono ridondanti. L'istruzione etichettata CLR0 posiziona lo stato di Carry a 0. L'istruzione etichettata FFAW saggia lo stato di Carry, e se rileva 0 salta all'istruzione che si trova più avanti etichettata FFA0. Ma poichè abbiamo già posizionato lo stato di Carry a 0, l'istruzione etichettata FFAW deve rilevare uno stato di Carry 0; perciò, il solo percorso logico permesso dopo un salto a CLR0 è un altro salto a FFA0. Perciò si possono sostituire le due istruzioni che saltano a CLR0 con istruzioni che saltano direttamente a FFA0; quindi si possono eliminare le istruzioni etichettate CLR0 e FFAW. Ciò elimina pure l'istruzione che salta a FFAW+2, poichè FFAW+2 indirizza un'istruzione BIT che diventa la successiva istruzione nella sequenza. Possiamo pure togliere l'istruzione SCF. Poichè si è tenuto conto delle condizioni Carry = 0 per i salti a FFA0, la mancanza è Carry = 1, che non deve più essere identificata. In tale modo, la nostra sequenza d'istruzioni può essere illustrata come segue:

```
        Sequenza vecchia                 Sequenza nuova
        IN      A,(2)                    IN      A,(2)
        -                                -
        -
        -                                -
        AND     22H                      AND     22H
        JR      Z,CLR0                   JR      Z,FFA0
        BIT     6,B                      BIT     6,B
        JR      Z,CLR0   )               JR      Z,FFA0
        SCF              |
        JR      FFAW+2   >     istruzioni
CLR0:   AND     A        |   non necessarie
FFAW:   JR      NC,FFA0  )
        BIT     5,A                      BIT     5,A
        -                                -
        -                                -
        -
```

Continuiamo la nostra analisi del programma con l'istruzione BIT 5,A.

Supponendo che CLR sia a 1, faremo poi un test su PW STROBE. A tale scopo, usiamo di nuovo un'istruzione BIT. PW STROBE è rappresentata dal bit 5 dell'Accumulatore.

Supponendo che PW STROBE sia 1, ci rimane da controllare la condizione di CH RDY. Per fare ciò eseguiamo di nuovo un'istruzione BIT; tuttavia questa volta noi saggiamo il contenuto del bit 1. Poichè l'istruzione BIT ha effetto solo sul flag dello stato Zero, possiamo eseguire tutte le istruzioni BIT di cui abbiamo bisogno sullo stesso byte senza cambiarlo.

Supponendo che si siano trovate tutte le condizioni per mettere "on" il flip-flop FFA, dobbiamo posizionare a 0 il bit 0 della Porta di I/O A1. Ciò è fatto mettendo il contenuto della Porta di I/O A0 nell'Accumulatore, resettando il bit appropriato, facendo tornare poi il risultato:

COMMUTAZIONE DI UN BIT NELLO STATO "ON"

Le ultime tre istruzioni della simulazione del flip-flop FFA sono tre istruzioni che posizionano a 1 il bit 0 della Porta di I/O A1 (riflettendo il fatto che il flip-flop FFA sia "off"). Queste tre istruzioni caricano il contenuto della Porta di I/O A1 nell'Accumulatore, posizionano il bit appropriato, quindi riportano il risultato:

Ora in tutta onestà, la sequenza di programma che abbiamo appena dèscritto è un modo ridicolo di simulare il flip-flop FFA e le sue tre porte associate.

E' ridicolo perchè abbiamo simulato ogni porta come una funzione di trasferimento indipendente. Invece consideriamo il flip-flop, con le sue tre porte, come una singola funzione di trasferimento. Possiamo rappresentare la funzione di trasferimento con la seguente definizione di stato:

Posizionare $\overline{Q}$ a 0 se RESET = 0, CH RDY = 1 e PW STROBE va da 0 a 1. Altrimenti posizionare $\overline{Q}$ a 1.

Come controlleremo che PW STROBE va da 0 a 1?

Usando interruzioni, il test sarebbe molto semplice, ma non useremo interruzioni fino al Capitolo 5.

CAMBIAMENTI DEI LIVELLI DEI SEGNALI RILEVATI SENZA INTERRUZIONI

Senza usare interruzioni c'è un solo modo di controllare una transizione da 0 a 1 di PW STROBE. Dobbiamo far entrare il contenuto della Porta di I/O B0 nell'Accumulatore, saggiare il bit 5 salvare il risultato, far entrare nuovamente il contenuto della Porta di I/O B0 nell'Accumulatore, saggiare il bit 5 di nuovo, quindi confrontare i due per un valore vecchio pari a 0 ed un valore nuovo pari a 1. Ma questo schema è rischioso; prenderà solo transizioni di segnale abbastanza fortunate da capitare tra due istruzioni che caricano il contenuto della Porta di I/O B0 nell'Accumulatore:

TEMPORIZZAZIONI DI EVENTI IN UN SISTEMA CON MICROCALCOLATORE

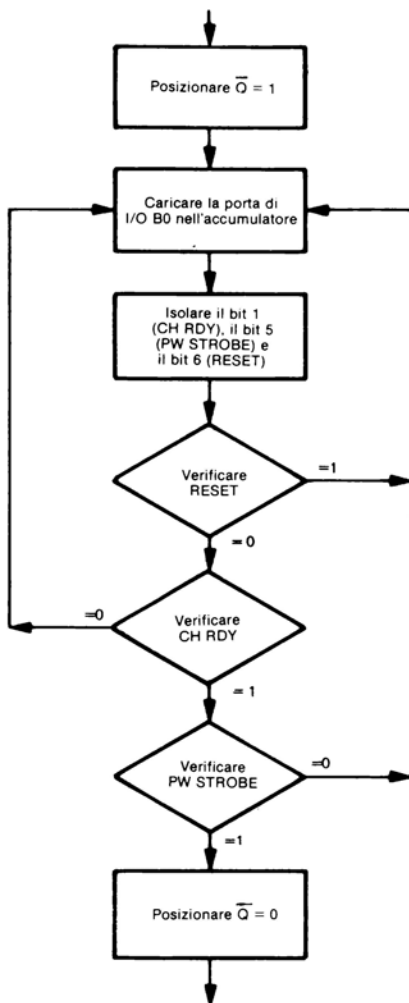
Nella logica di un programma su un microcalcolatore, tuttavia, non abbiamo bisogno di fare affidamento sulle transizioni del segnale. Una sequenza di esecuzione di istruzioni determina sequenze di eventi. Il concetto di temporizzazione sul fronte di salita di un impulso di segnale non ha significato. Invece di usare le transizioni del segnale PW STROBE, useremo, perciò, i livelli del segnale PW STROBE. Il flip-flop FFA può essere ora descritto con la seguente definizione di stato:

Posizionare $\overline{Q}$ a 0 se RESET = 0, CH RDY = 1 e PW STROBE = 1. Altrimenti posizionare $\overline{Q}$ a 1.

TEMPORIZZAZIONE E SEQUENZA LOGICA

Se siete progettisti logici, potete essere profondamente infastiditi dal modo gaio con cui abbiamo semplicemente sostituito un riferimento ai fronti con un riferimento ai livelli. Ciò può essere fatto in sistema a microcalcolatore perchè la programmazione di un microcalcolatore ci dà un ulteriore grado di libertà, in confronto col progetto di logica digitale; l'ordine col quale voi

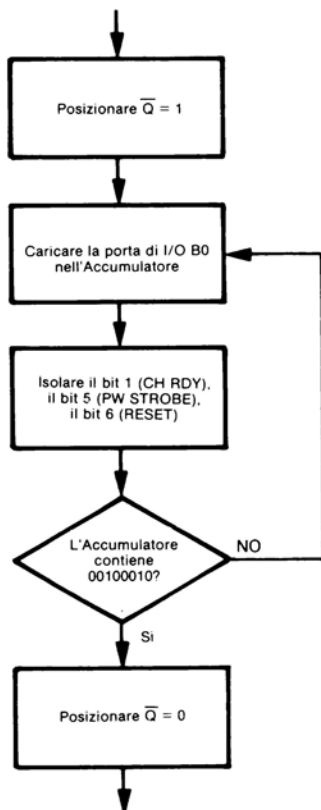
riempite di componenti logici una piastra PC non ha niente a che fare con la sequenza nella quale capitano eventi logici. La sequenza logica sarà controllata da riferimenti a fronti e a livelli. Ma l'ordine col quale scrivete istruzioni in linguaggio assembly è l'ordine nel quale saranno eseguite le istruzioni.

Per comprendere questo punto, guardiamo il seguente flowchart che rappresenta la definizione di stato per il flip-flop FFA:

Nuovamente, un rettangolo rappresenta una operazione di manipolazione o di movimenti di dati, e ogni rombo rappresenta della logica che saggia la condizione del flag di stato.

L'ordine nel quale scrivete le istruzioni è l'ordine col quale le istruzioni saranno eseguite. Con riferimento alla flowchart precedente, questa sequenza d'esecuzione è

rappresentata dalla linea continua di una freccia che punta verso il basso. Speciali istruzioni di Salto Condizionato permettono di modificare la normale sequenza, come rappresentato dalle frecce orizzontali che escono dai lati dei rombi. Potete seguire le frecce fino al punto dove vi conduce l'istruzione di Salto Condizionato.

Riscriveremo ora la simulazione del flip-flop FFA trattando il flip-flop e le tre porte logiche di CLR come una singola funzione di trasferimento.

Poichè RESET, CH RDY e PW STROBE sono tutti collegati ai pin della Porta di I/O B0, carichiamo il contenuto della Porta di I/O B0 nell'Accumulatore ed isoliamo tutti e tre i bit. Ora c'è solo una combinazione di valori che questi tre bit possono avere se deve cominciare un nuovo ciclo di stampa. RESET deve essere uguale a 0, mentre CH RDY e PW STROBE devono essere entrambi uguali ad 1. Ritracceremo quindi il flowchart del programma come segue:

La nostra sequenza d'istruzioni si condensa alle poche istruzioni seguenti:

```
; Simulazione di FFAW e della logica associata
        IN      A,(4)           ; Posizionamento iniziale ad 1 del bit 0 della Porta
                                ; di I/O A1
```

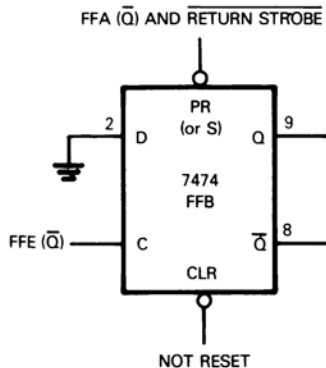
```
          SET    0,A
          OUT    (4),A
; Carica il contenuto della Porta di I/O B0 nell'Accumulatore ed isola i bit 1, 5 e 6
; per CH RDY, PW STROBE e RESET rispettivamente
FFAW:     IN     A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
          AND    62H            ; isola i bit 6, 5 e 1, se RESET = 0
          CP     22H            ; CH RDY = 1 e PW STROBE = 1, parte un nuovo
                                ; ciclo di stampa
          JR     NZ,FFAW        ; Altrimenti ritorna a FFAW
          IN     A,(4)          ; Parte un nuovo ciclo di stampa posizionando A0
          RES    0,A            ; Il bit 0 della porta di I/O A1
          OUT    (4),A
; La sequenza d'istruzioni del nuovo ciclo di stampa comincia qui
```

Le prime tre istruzioni nella sequenza precedente posizionano semplicemente a 1 il bit 0 della Porta di I/O A1. Questa è una anticipazione che non sta cominciando un nuovo ciclo di stampa. Quattro istruzioni a cominciare dall'istruzione etichettata FFAW è tutto ciò che occorre per verificare le condizioni che danno l'avvio ad un nuovo ciclo di stampa. Queste quattro istruzioni sono eseguite in 36 cicli di clock; supponendo un clock di 500 nanosecondi, ciò significa che l'impulso PW STROBE deve essere alto per almeno 18 microsecondi.

Fornendo un RESET uguale a 0 mentre CH RDY e PW STROBE sono uguali a 1, deve cominciare un nuovo ciclo di stampa, così le ultime tre istruzioni posizionano a 0 il bit 0 della Porta di I/O A1.

La nostra simulazioni del flip-flop FFA è completata.

FLIP-FLOP FFB_W

Il prossimo dispositivo nella nostra sequenza logica è un altro flip-flop 7474, indicato con FFB_W nella Figura 3-1; è proprio sulla destra di FFA_W. Questo flip-flop può essere illustrato come segue:

La tabella di funzionamento che segue descrive FFB, come collegato sopra, con il suo ingresso D collegato a 0.

FFA ($\overline{Q}$)	$\overline{\text{RETURN STROBE}}$	PRESET	NOT RESET (CLR)	FFE ($\overline{Q}$) =CLOCK	Q	$\overline{Q}$
0	0	0	1	X	1	0
0	1	0	0	X	instabile	
1	0	0				
1	1	1	0	X	0	1
		1	1	0→1	0	1

Il Capitolo 2 fornisce la tabella di funzionamento standard del flip-flop 7474; tutto ciò che abbiamo fatto è di aver tolto la colonna D e le righe che mostrano D = 1. Possiamo pure togliere la colonna CLR e tutte le righe che mostrano CLR = 0, poichè CLR è attaccato a NOT RESET. NOT RESET sarà sempre 1 in un ciclo di stampa, poichè FFA non andrà "on" se NOT RESET è 0.

La seguente tabella semplificata di funzionamento può essere ora usata per FFB, supponendo che CLR (NOT RESET) sarà sempre 1 e che D sarà sempre 0:

FFA ($\overline{Q}$) AND $\overline{\text{RETURN STROBE}}$ =PRESET	FFE ($\overline{Q}$) =CLOCK	Q	$\overline{Q}$
0	0 or 1	1	0
1	0→1	0	1

Diamo un'occhiata all'ingresso di PRESET di FFB; esso è FFA ($\overline{Q}$) in AND con $\overline{\text{RETURN STROBE}}$.

CICLO DI STAMPA DI RIPOSIZIONAMENTO DELLA RUOTA DI STAMPA

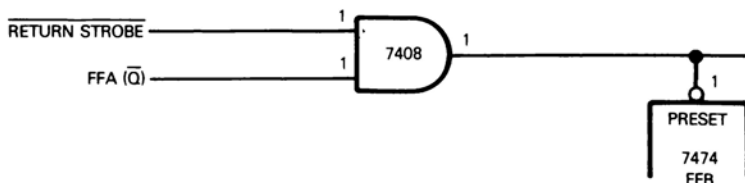
Ricordiamo che $\overline{\text{RETURN STROBE}}$ è un segnale d'ingresso della logica esterna per far iniziare uno speciale ciclo di stampa che riporta la ruota di stampa nella sua posizione di visibilità, ma non alimenta il martelletto di stampa o non stampa nessun carattere. Chiamiamo ciò un ciclo di stampa di "Riposizionamento della ruota di stampa". Quindi $\overline{\text{RETURN STROBE}}$ deve essere alto tra cicli di stampa.

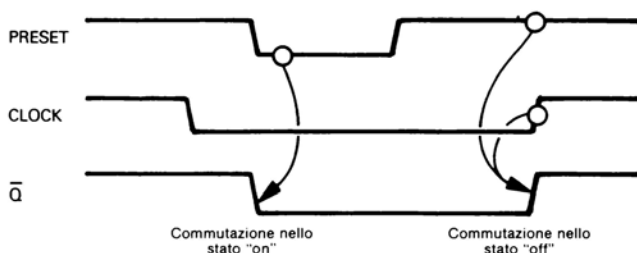
Poichè $\overline{\text{RETURN STROBE}}$ è basso in ingresso come metodo alternativo per iniziare un ciclo di stampa, quando si simula FFB, dobbiamo considerare $\overline{\text{RETURN STROBE}}$ in due modi:

1) **Come partecipe dell'ingresso di PRESET.**
2) **Come segnale che può dar inizio ad un ciclo di stampa, bypassando il flip-flop FFA.**

Ma dapprima definiamo la condizione del flip-flop FFB tra due cicli di stampa.

Come già abbiamo visto nella simulazione del flip-flop FFA, l'uscita di FFA ($\overline{Q}$) è alta finchè non inizia un ciclo di stampa, allorchè $\overline{Q}$ va basso; perciò l'uscita di FFA ($\overline{Q}$) è alta tra due cicli di stampa. Per definizione $\overline{\text{RETURN STROBE}}$ è alto tra due cicli di stampa, poichè si usa $\overline{\text{RETURN STROBE}}$ basso per iniziare un ciclo di stampa di riposizionamento della ruota di stampa.

Quindi l'ingresso di PRESET di FFB sarà alto tra due cicli di stampa:

Poichè PRESET è un ingresso alto tra due cicli di stampa, supporremo che all'inizio di ogni ciclo di stampa FFB sia off; cioè l'uscita Q bassa e l'uscita Q̄ alta. Ciò presuppone pure che alcuni istanti prima PRESET sia stato messo alto quando l'uscita Q̄ del flip-flop FFE è andata da 0 ad 1. Come vedrete più tardi, ciò è quanto accade alla fine di ogni ciclo di stampa.

In un nuovo ciclo di stampa, perciò, FFB ha un ingresso di PRESET alto, con l'uscita Q alta e con l'uscita Q̄ bassa. Questo flip-flop agisce ora come un interruttore: è messo "on" quando l'ingresso di PRESET è basso; successivamente è messo "off" da una transizione del clock da 0 a 1 che avviene dopo che PRESET è nuovamente andato alto:

Il posizionamento nello stato di "on" illustrato sopra avviene in due circostanze:

1) Immediatamente dopo l'inizio di un nuovo ciclo di stampa, quando FFA fa uscire Q̄ basso, forzando così il PRESET basso.
2) Quando $\overline{\text{RETURN STROBE}}$ è basso in ingresso segnalando un ciclo di stampa di riposizionamento della ruota di stampa.

Il posizionamento nello stato di "off" avviene quando l'uscita (Q̄) di FFE fa una transizione da basso ad alto mentre si mantiene alto l'ingresso di PRESET; ciò accade alla fine di ogni ciclo di stampa.

SIMULAZIONE DEL FLIP-FLOP FFB

COMMUTAZIONE DEI BIT NELLO STATO "ON"

Il bit 1 della Porta di I/O A1 è stato assegnato all'uscita Q̄ del flip-flop FFB. La commutazione nello stato "on" illustrata sopra è quindi simulata dalle seguenti tre istruzioni:

```
IN    A,(4)     ; Carica il byte del dato del flip-flop
RES   1,A       ; Posiziona a 0 il bit 1
OUT   (4),A     ; Rimemorizza il byte dati del flip-flop
```

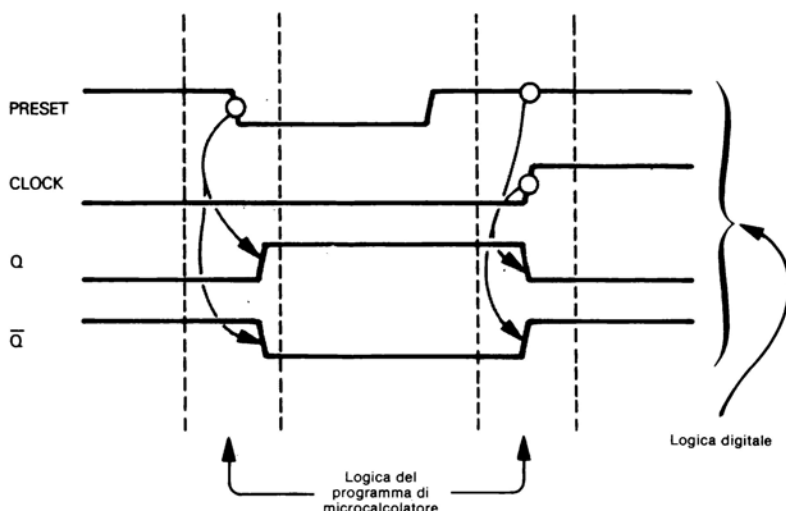
COMMUTAZIONE DEI BIT NELLO STATO "OFF"

Successivamente si può illustrare come segue la commutazione nello stato "off":

```
IN    A,(4)       ; Carica il byte dati del flip-flop
SET   1,A         ; Posiziona a 1 il bit 1
OUT   (4),A       ; Rimemorizza il byte dati del flip-flop
```

Ora incontriamo una situazione dove, con tutta la buona volontà, non saremo capaci di simulare direttamente la nostra logica digitale.

E' abbastanza facile disegnare un flip-flop 7474 in un diagramma logico e collegare i suoi pin ai segnali opportuni. Dopo aver fatto ciò, non dovete più preoccuparvi se un segnale cambia stato o no.

Sfortunatamente una sequenza di istruzioni in linguaggio assembly non ha nè pin nè segnali; **un linguaggio assembly simulerà solo eventi che accadono in un certo istante di tempo. Per il flip-flop FFB, ciò può essere illustrato come segue:**

Immediatamente dopo che il flip-flop FFA va "on" per introdurre un nuovo ciclo di stampa, esso mette la sua uscita $\overline{Q}$ bassa, che a sua volta mette "on" il flip-flop FFB. FFB non sarà messo "off" se non molto più tardi nel ciclo di stampa, quando l'uscita $\overline{Q}$ di FFE sarà alta. **Dobbiamo perciò dividere la simulazione di FFB in due parti:**

1) All'inizio del nostro programma simuleremo la commutazione nello stato "on" di FFB, poichè cronologicamente esso è il successivo evento nel ciclo di stampa.
2) Più avanti nel programma, quando si simulerà il posizionamento $\overline{Q}$ alto di FFE, dobbiamo ricordarci di simulare la commutazione nello stato "off" di FFB.

Ma ciò non è tutto la simulazione di FFB. **Dobbiamo pure modificare la sequenza di istruzioni che si esegue tra due cicli di stampa, così che si possa simulare l'ingresso RETURN STROBE messo basso per inizializzare un ciclo di riposizionamento della ruota di stampa.**

Ecco come è ora il nostro programma, dove le istruzioni nuove o modificate sono ombreggiate:

```
; Esecuzione del programma tra due cicli di stampa
; Posizionamento iniziale a 1 dei bit 1 e 0 della Porta di I/O A1
        IN    A,(4)        ; La Porta di I/O A1 entra nell'Accumulatore
        OR    3            ; Posizionamento dei bit 1 e 0
        OUT   (4),A        ; Ritorno del risultato
; Test per RETURN STROBE basso
STBHI:  IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
        BIT   4,A          ; Test sul bit di RETURN STROBE
        JR    Z,FFB        ; Se è 0 saltare alla simulazione di FFB
; Simulazione di FFAW e della logica associata
; Carica il contenuto della Porta di I/O B0 nell'Accumulatore ed isola i bit 1, 5 e 6
; rispettivamente per CH RDY, PW STROBE e RESET
        IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
        AND   62H          ; Isola i bit 6, 5 e 1. Se RESET = 0
        CP    22H          ; CH RDY = 1 e PW STROBE = 1, comincia un nuo-
                           ; vo ciclo di stampa
        JR    NZ,STBHI     ; Altrimenti si ritorna a STBHI
        IN    A,(4)        ; Inizio di un nuovo ciclo di stampa posizionando a 0
        RES   0,A          ; Il bit 0 della Porta di I/O A1
        OUT   (4),A
; La sequenza di istruzioni per il nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB
FFB:    IN    A,(4)        ; Carica la Porta di I/O A1 nell'Accumulatore
        RES   1,A          ; Posiziona a 0 il bit 1
        OUT   (4),A        ; Memorizza il risultato
```

Non abbiamo completamente perfezionato la nostra simulazione del flip-flop FFB. Osserviamo che l'uscita $\overline{Q}$ da FFB va a:

1) Una porta AND 7411, dislocata approssimativamente in coordinate B5.

2) Una porta OR 7432, dislocata in C7.

L'uscita ($\overline{Q}$) di FFB non è affatto inutile, ma la vedremo più tardi.

Dapprima consideriamo la porta AND 7411 dislocata in B5.

Se vi riferite alla descrizione dei segnali di uscita, noterete che CH RDY era stato dichiarato alto tra due cicli di stampa, ma basso durante un ciclo di stampa.

In realtà, CH RDY esce dalla porta AND 7411 dislocata in B5; quindi, tra due cicli di stampa, tutti i tre ingressi della porta AND devono essere alti. La nostra analisi del flip-flop FFB mostra che l'uscita $\overline{Q}$ sarà realmente alta tra due cicli di stampa, ma per ora dovete credere che gli altri due segnali d'ingresso della porta AND saranno anch'essi alti tra due cicli di stampa.

In ogni caso, non appena il flip-flop FFB va "on", la sua uscita $\overline{Q}$ va bassa, significando che qualunque cosa stiano facendo gli altri due ingressi della porta AND 7411, CH RDY andrà pure basso. Questo cambiamento dello stato di CH RDY è simulato aggiungendo le istruzioni seguenti al nostro programma:

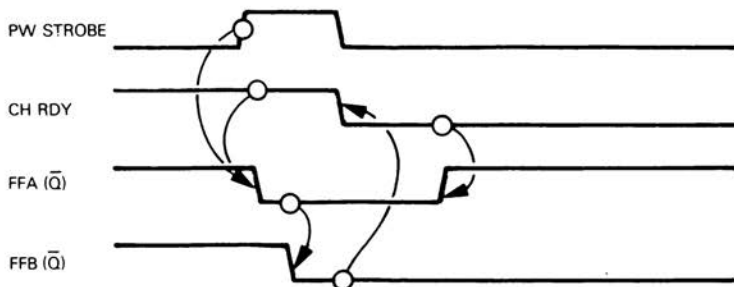
```
; Test su RETURN STROBE basso
STBHI:  IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
        BIT   4,A          ; Test sul bit RETURN STROBE
        JR    Z,FFB        ; Se è 0, saltare alla simulazione di FFB
; Simulazione di FFAW e della logica associata
; Carica il contenuto della Porta di I/O B0 nell'Accumulatore e isola i bit 1, 5 e 6
```

```
; rispettivamente per CH RDY, PW STROBE e RESET
          IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
          AND   62H          ; Isola i bit, 6,5 e 1. Se RESET = 0
          CP    22H          ; CH RDY = 1 e PW STROBE = 1, parte il nuovo ci-
                             ; clo di stampa
          JR    NZ,STBHI     ; Altrimenti si ritorna a STBHI
          IN    A,(4)        ; Inizio del nuovo ciclo di stampa posizionando a 0
          RES   0,A          ; Il bit 0 della Porta di I/O A1
          OUT   (4),A
; La sequenza di istruzioni per il nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB
FFB:      IN    A,(4)        ; Carica la Porta di I/O A1 nell'Accumulatore
          RES   1,A          ; Posiziona a 0 il bit 1
          OUT   (4),A        ; Memorizza il risultato
; Simulazione della porta AND 7411 mettendo basso CH RDY
          IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
          RES   1,A          ; Posiziona il bit 1 a 0
          OUT   (2),A        ; Memorizza il risultato
```

Affrontiamo ora un problema interessante. CH RDY diventa l'ingresso D del flip-flop FFA e contribuisce all'ingresso CLR di FFA. **Cosa succede quando CH RDY va basso in risposta alla commutazione nello stato "on" di FFB?**

Notate che PW STROBE è un impulso alto, perciò la porta OR dislocata in coordinate B2 si affida a CH RDY alto per fornire un ingresso alto alla porta AND seguente.

Questa porta AND*, a sua volta, fornisce un ingresso alto al CLR di FFA. In altre parole, dall'istante in cui il flip-flop FFB va "on" e mette CH RDY basso, PW STROBE dovrà essere già andato basso; così entrambi gli ingressi PW STROBE e CH RDY saranno bassi. **Se riguardate la tabella della verità di CLR del flip-flop FFA, troverete che quando sia CH RDY che PW STROBE sono 0, CLR sarà sempre 0.**

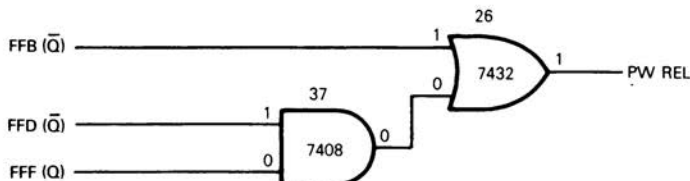
Quindi il flip-flop FFA sarà messo "off":

Cosa significa ciò? La nostra conclusione è che il flip-flop FFA mette sè stesso "on" all'inizio di un ciclo di stampa, ma vi rimane solo abbastanza da mettere "on" il flip-flop FFB. Quando FFB va "on", esso posiziona CH RDY basso, e ciò mette "off" il flip-flop FFA.

TEMPORIZZAZIONE E SEQUENZA LOGICA

Ma ecco la difficoltà: se riguardate la Figura 3-1, troverete che il flip-flop FFA collabora alla generazione dell'ingresso J del flip-flop FFC, oltre che far commutare il flip-flop FFB.

Ora che gli eventi sono stati serializzati nel tempo, possiamo andare avanti e simulare la commutazione nello stato "off" del flip-flop FFA, dal momento che vedremo, quando simuleremo FFC, che questi riceve $\overline{Q}$ basso del flip-flop FFA. Tenendo in mente questa precauzione, stenderemo il nostro programma come segue:

```
; Test su RETURN STROBE basso
STBHI:  IN   A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        BIT  4,A        ; Test sul bit RETURN STROBE
        JR   Z,FFB      ; Se è 0, saltare alla simulazione di FFB
; Simulazione di FFAW e della logica associata
; Carica il contenuto della Porta di I/O B0 nell'Accumulatore e isola i bit 1, 5 e 6
; rispettivamente per CH RDY, PW STROBE e RESET
        IN   A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        AND  62H        ; Isola i bit 6, 5 e 1. Se RESET = 0
        CP   22H        ; CH RDY = 1 e PW STROBE = 1, comincia un nuo-
                        ; vo ciclo di stampa
        JR   NZ,STBHI   ; Altrimenti ritorna a STBHI
        IN   A,(4)      ; Inizio di un nuovo ciclo di stampa posizionando a 0
        RES  0,A        ; Il bit 0 della Porta di I/O A1
        OUT  (4),A
; La sequenza di istruzioni del nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB
FFB:    IN   A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        RES  1,A        ; Posiziona a 0 il bit 1
        OUT  (4),A      ; Memorizza il risultato
; Simulazione della porta AND 7411 mettendo basso CH RDY
        IN   A,(2)      ; La porta di I/O B0 entra nell'Accumulatore
        RES  1,A        ; Posiziona il bit 1 a 0
        OUT  (2),A      ; Memorizza il risultato
; CH RDY basso mette FFA "off". Posiziona a 1 il bit 0 della Porta di I/O A1
        IN   A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        SET  0,A        ; Posiziona ad 1 il bit 0
        OUT  (4),A      ; Memorizza il risultato
```

Guardiamo ora la porta OR dislocata in coordinate C7. Questa porta riceve l'uscita $\overline{Q}$ del flip-flop FFB come ingresso per generare PW REL. L'altro ingresso di questa porta OR è l'AND dell'uscita Q del flip-flop FFF con l'uscita $\overline{Q}$ del flip-flop FFD. Troverete subito che questi flip-flop sono messi "off" tra due cicli di stampa; essi sono messi "on" in sequenza durante il corso del ciclo di stampa. Quando FFB va "on", FFF andrà "off", cioè la sua uscita sarà bassa; in tale modo la porta AND dislocata in C6 avrà l'uscita bassa, il che significa che **la porta OR 26 fa affidamento sull'uscita alta di $\overline{Q}$ di FFB per porre alto PW REL in uscita:**

Ora, quando FFB va "on" e mette $\overline{Q}$ basso, PW REL sarà pure basso in uscita. Dobbiamo perciò modificare il nostro programma per fare uscire bassi i bit 0 e 1 della Porta di I/O B0, poichè sia PW REL che CH RDY stanno per andare bassi. Ecco come

è ora il nostro programma:

```
; Test su RETURN STROBE basso
STBHI:  IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        BIT   4,A         ; Test sul bit RETURN STROBE
        JR    Z,FFB       ; Se è 0, saltare alla simulazione di FFB
; Simulazione di FFAW e della logica associata
; Carica il contenuto della Porta di I/O B0 nell'Accumulatore e isola i bit 1, 5 e 6
; rispettivamente per CH RDY, PW STROBE e RESET
        IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        AND   62H         ; Isola i bit 6, 5 e 1. Se RESET = 0
        CP    22H         ; CH RDY = 1 e PW STROBE = 1, parte il nuovo ci-
                          ; clo di stampa
        JR    NZ,STBHI    ; Altrimenti si ritona a STBHI
        IN    A,(4)       ; Inizio del nuovo ciclo di stampa posizionando a 0
        RES   0,A         ; Il bit 0 della Porta di I/O A1
        OUT   (4),A
; La sequenza di istruzioni per il nuovo ciclo di stampa comincia qui
; Simula la commutazione "on" del flip-flop FFB
FFB:    IN    A,(4)       ; La Porta di I/O A1 entra nell'Accumulatore
        RES   1,A         ; Riposizionamento a 0 del bit 1
        OUT   (4),A       ; Memorizza il risultato
; Simulazione della porta AND 7411 mettendo basso CH RDY. Anche la porta OR
; 7432 mette basso PW REL
        IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        AND   0FCH        ; Posiziona a 0 i bit 0 e 1
        OUT   (2),A       ; Memorizza il risultato
; CH RDY basso mette FFA "off". Posiziona a 1 il bit 0 della Porta di I/O A1
        IN    A,(4)       ; Carica la Porta di I/O A1 nell'Accumulatore
        SET   0,A         ; Posiziona a 1 il bit 0
        OUT   (4),A       ; Memorizza il risultato
```

Dobbiamo fare qualcosa sull'uscita Q del flip-flop FFB? Se guardate questa uscita vedrete che è collegata direttamente agli ingressi di RESET dei flip-flop FFC, FFD e FFE. Inoltre essa diventa uno degli ingressi del multivibratore 555.

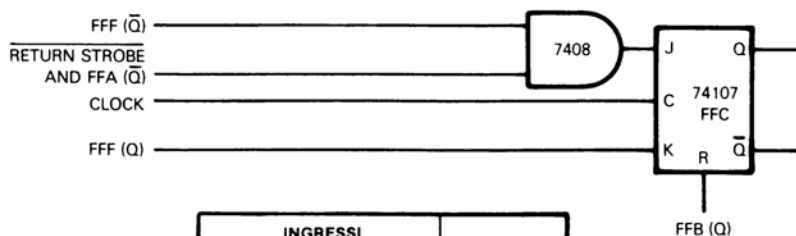
In effetti, l'uscita Q di FFB è un segnale "che tiene fermo"; quando è basso tiene "off" i quattro dispositivi ad esso collegati, e quando è alto questi quattro dispositivi sono commutati nello stato "on".

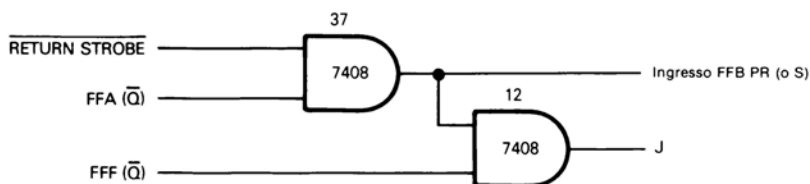
L'uscita Q di FFB sarà considerata quando simuleremo i quattro dispositivi collegati a questo segnale. Abbiamo perciò, eseguito la simulazione del flip-flop FFB.

FLIP-FLOP FFC

Questo è il flip-flop 74107 di coordinate C2 in Figura 3-1. Poichè si simuleranno i quattro flip-flop 74107, si dovrà far riferimento al Capitolo 2, se non si ricordano immediatamente le caratteristiche di questo dispositivo.

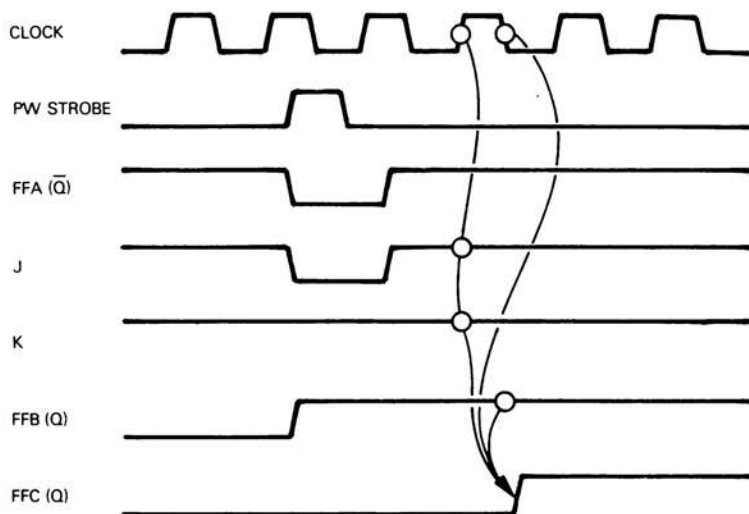
Isoliamo il flip-flop FFC per vedere come funziona.

Tra due cicli di stampa, l'uscita Q di FFB, poichè è bassa, mette il flip-flop FFC nello stato "off". Quindi FFC ha le uscite Q bassa e $\overline{Q}$ alta.
Ciò che accade quando si fa commutare FFB, dipende dagli ingressi J e K che si trovano su FFC.

INGRESSI					
R	C	J	K	Q	$\overline{Q}$
L	X	X	X	L	H
H	⎍	L	L	rimane lo stesso	
H	⎍	H	L	H	L
H	⎍	L	H	L	H
H	⎍	H	H	s'inverte	

Tra due cicli di stampa si mette "off" il flip-flop FFF, perciò la sua uscita Q sarà bassa. FFC riceve il suo ingresso K dall'uscita Q di FFF, quindi quando FFC commuta nello stato "on", il suo ingresso K sarà 0.
L'ingresso J di FFC è generato come segue:

FFF ($\overline{Q}$) sarà alto, poichè FFF è commutato nello stato "off". L'ingresso J di FFC sarà perciò identico all'ingresso PR di FFB, che abbiamo già descritto.

Riassumendo, la sequenza dei segnali che mette FFC nello stato "on" è illustrata a pagina 3-45.

Quando l'uscita Q di FFB va alta, non vincolando FFC, FFC aspetta finchè l'uscita $\overline{Q}$ di FFA va nuovamente alta; allora FFC riceverà un ingresso alto su J e un ingresso basso su K. Q sarà messo alto in uscita e $\overline{Q}$ sarà messo basso sui fronti dell'impulso di clock di FFC.

FFC aspetta l'uscita $\overline{Q}$ di FFA per andare nuovamente alto, perchè mentre FFA è "on" la sua uscita $\overline{Q}$ è bassa. Mentre l'uscita ($\overline{Q}$) di FFA (o $\overline{\text{RETURN STROBE}}$)

ha un impulso basso, FFC riceve un basso sull'ingresso J. Poichè gli ingressi J e K di FFC sono bassi, le sue uscite non cambieranno — ciò è una delle proprietà del flip-flop 74107.

Il flip-flop FFC rimarrà nel suo stato "on" durante il ciclo di stampa finchè il flip-flop FFF non commuta nello stato "on". In quell'istante, il flip-flop FFC riceverà un alto sull'ingresso K e un basso sull'ingresso J; e ciò provocherà la commutazione di FFC nello stato "off".

SIMULAZIONE DEL FLIP-FLOP FFC

La simulazione del flip-flop FFC è, in realtà, diretta; essa implica questi tre passi:

1) Dobbiamo adattare le istruzioni d'inizializzazione per assicurarci che il flip-flop FFC sia rimesso nello stato "off" tra due cicli di stampa.
2) La simulazione del flip-flop FFB deve essere seguita immediatamente da istruzioni che simulino la commutazione nello stato "on" del flip-flop FFC.
3) Dobbiamo ricordarci di simulare la commutazione nello stato "off" di FFC — ma ciò non accadrà se non più avanti nel programma.

Ora le seguenti modifiche all'inizio del programma assicurano la simulazione detta commutazione nello stato "off" del flip-flop FFC tra due cicli di stampa:

```
; Esecuzione del programma tra due cicli di stampa
; Inizialmente si posizionano a 1 il bit 1 e 0 della Porta di I/O A1, a 0 il bit 2
        IN    A,(4)      ; La Porta di I/O A1 entra nell'Accumulatore
        OR    3          ; Posizionamento dei bit 1 e 0
        RES   2,A        ; Posizionamento a 0 del bit 2
        OUT   (4),A      ; Ritorno del risultato
; Test su RETURN STROBE basso
STBHI:  IN    A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        BIT   4,A        ; Test sul bit di RETURN STROBE
        JR    Z,FFB      ; Se esso è 0, saltare alla simulazione di FFB
```

Tutto ciò che abbiamo fatto è stato di aggiungere l'istruzione RES per posizionare a 0 il bit 2 della Porta di I/O A1:

```
                         Contenuto
                      dell'Accumulatore A

                         76543210 <---- Bit. n.
IN        A,(4)          XXXXXXXX
OR        3              00000011
                         --------
                         XXXXXX11
RES       2,A            XXXXX011
```

Ricordiamo che il bit 2 della Porta di I/O A1 è stato assegnato al flip-flop FFC.

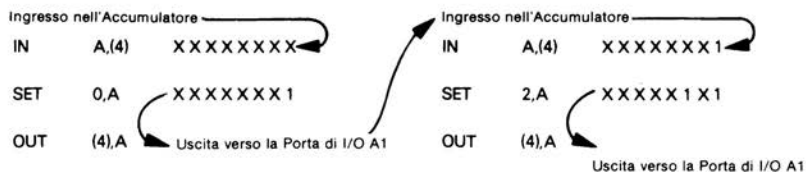
TEMPORIZZAZIONE E SEQUENZA LOGICA

Che cosa si può dire sul ritardo di tempo che separa la commutazione nello stato "on" dei flip-flop B e C? Ricordiamo che il flip-flop FFC non andrà "on" finchè il flip-flop FFB non avrà messo "off" il flip-flop FFA. Se questo è un ciclo di stampa di riposizionamento della ruota di stampa, allora FFC non andrà "on" finchè l'ingresso $\overline{\text{RETURN STROBE}}$ è ancora alto.

La semplicità o la complessità del nostro problema di temporizzazione dipende interamente dalla logica della Figura 3-1. Non c'è niente nella logica di Figura 3-1 che richieda un ritardo di tempo di durata fissata o, in questo caso, un ritardo di tempo che separi la commutazione nello stato "on" di FFB e di FFC. Non presteremo perciò attenzione alle considerazioni di temporizzazione associate alla commutazione nello stato "on" di FFC; piuttosto, aggiungeremo la simulazione alla fine del nostro programma come segue:

```
; La sequenza di istruzioni del nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB
FFB:    IN   A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        RES  1,A        ; Posiziona il bit 1 a 0
        OUT  (4),A      ; Memorizza il risultato
; Simulazione della porta AND 7411 mettendo basso CH RDY. Anche la porta OR
; 7432 mette basso PW REL
        IN   A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        AND  0FCH       ; Posiziona a 0 i bit 0 ed 1
        OUT  (2),A      ; Memorizza il risultato
; CH RDY basso mette FFA "off". Posiziona ad 1 il bit 0 della Porta di I/O A1
(A)     IN   A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        SET  0,A        ; Posiziona a 1 il bit 0
        OUT  (4),A      ; Memorizza il risultato
; Simulazione della commutazione nello stato "on" del flip-flop 74107 FFC.
; Posizionare ad 1 il bit 2 della Porta di I/O A1
        IN   A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        SET  2,A        ; Posiziona il bit 2 ad 1
        OUT  (4),A      ; Memorizza il risultato
```

PROGRAMMI RESI PIU' BREVI

Se cominciate a pensare come un programmatore, rileverete un'opportunità di economia nella simulazione della commutazione nello stato "on" del flip-flop FFC. **Osservate che le tre istruzioni direttamente sopra ad (A) posizionano pure a 1 un bit della Porta di I/O A1.** Ciò genera la seguente sequenza di eventi:

Possiamo combinare le due operazioni come segue:

Le istruzioni segnate da Ⓐ ora scompaiono e sono sostituite da queste modifiche segnate da Ⓑ :

```
; La sequenza di istruzioni del nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB
FFB:    IN   A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        RES  1,A        ; Posiziona il bit 1 a 0
        OUT  (4),A      ; Memorizza il risultato
; Simulazione della porta AND 7411 mettendo basso CH RDY. Anche la porta OR
; 7432 mette basso PW REL
        IN   A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        AND  0FCH       ; Posiziona a 0 i bit 0 ed 1
        OUT  (2),A      ; Memorizza il risultato
; CH RDY basso mette FFA "off". Posiziona a 1 il bit 0 della Porta di I/O A1
; Simula pure la commutazione nello stato "on" di FFC. Posiziona a 1 il bit 2 della
; Porta di I/O A1
        IN   A,(4)      ; Carica la Porta di I/O A1 nell'Accumulatore
        OR   5          ; Posiziona i bit 2 e 0 ad 1
(B)     OUT  (4),A      ; Memorizza il risultato
```

SIMULAZIONE DELL'IMPULSO DI AVVIO DEL MOTO DEL NASTRO (START RIBBON MOTION PULSE)

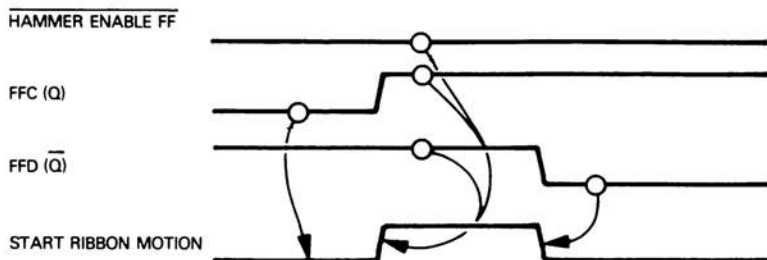
Ricordiamo che, in un ciclo di stampa, si dà subito un impulso alto al segnale di uscita START RIBBON MOTION per dare l'avvio alla logica esterna che fa avanzare il nastro; così, quando si alimenta il martelletto di stampa, di fronte al carattere che deve essere stampato, c'è del nastro fresco.

Il segnale START RIBBON MOTION è generato da una porta AND 7411 (numero 7) dislocata in coordinate D6 nella Figura 3-1. Questa porta AND ha tre ingressi:

1) $\overline{\text{HAMMER ENABLE FF}}$. Questo è un segnale d'ingresso che identifica un ciclo di stampa di riposizionamento della ruota di stampa.
2) L'uscita Q del flip-flop FFC.
3) L'uscita $\overline{\text{Q}}$ del flip-flop FFD.

$\overline{\text{HAMMER ENABLE FF}}$ sarà alto senza che sia in corso un ciclo di stampa di riposizionamento della ruota di stampa, nel qual caso il nastro non deve essere mosso. Questo segnale sopprime perciò l'impulso START ENABLE MOTION.

Tra due cicli di stampa, i flip-flop FFC e FFD sono entrambi nello stato "off"; perciò (Q) di FFC è bassa e ($\overline{\text{Q}}$) di FFD è alta. **L'uscita (Q) di FFC mantiene basso il segnale START ENABLE MOTION.**

Quando FFC va "on" durante un normale ciclo di stampa tutti gli ingressi della porta AND 7 saranno alti, così l'impulso START ENABLE MOTION sarà alto; esso rimarrà alto finchè il flip-flop FFD non va "on", nel quale istante FFD avrà l'uscita $\overline{Q}$ bassa; ciò farà andare basso l'impulso START ENABLE MOTION. **La temporizzazione può essere illustrata come segue:**

Se guardate il diagramma della temporizzazione illustrato in Figura 3-2, vedrete che l'impulso di uscita del segnale START ENABLE MOTION è estremamente breve. Perciò, invece di usare il flip-flop FFD per temporizzare la fine dell'impulso START ENABLE MOTION, eseguiremo semplicemente le istruzioni per mettere "on" il bit 3 della Porta di I/O B0, e per rimetterlo subito dopo "off", come segue:

```
; La sequenza di istruzioni del nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB
FFB:    IN    A,(4)       ; Carica la Porta di I/O A1 nell'Accumulatore
        RES   1,A         ; Posiziona il bit 1 a 0
        OUT   (4) A       ; Memorizza il risultato
; Simulazione della porta AND 7411 mettendo basso CH RDY. Anche la porta OR
; 7432 mette basso PW REL
        IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        AND   0FCH        ; Posiziona a 0 i bit 0 e 1
        OUT   (2),A       ; Memorizza il risultato
; CH RDY basso mette FFA "off". Posiziona a 1 il bit 0 della Porta di I/O A1
; Simula pure la commutazione nello stato "on" di FFC. Posiziona a 1 il bit 2 della
; Porta di I/O A1
        IN    A,(4)       ; Carica la Porta di I/O A1 nell'Accumulatore
        OR    5           ; Posiziona i bit 2 e 0 ad 1
        OUT   (4),A       ; Memorizza il risultato
; Impulso START ENABLE MOTION alto
        IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        SET   3,A         ; Posiziona alto il bit 3
        OUT   (2),A       ; Uscita verso la Porta di I/O B0
        RES   3,A         ; Posiziona basso il bit 3
        OUT   (2),A       ; Uscita verso la Porta di I/O B0
```

CALCOLO DELLA DURATA DELL'IMPULSO

Possiamo calcolare la durata dell'impulso START RIB MOTION sommando i tempi di esecuzione delle istruzioni che posizionano il pin 3 della Porta di I/O B prima alto poi basso:

```
Cicli         Istruzione
 11      OUT     (2),A       ; Uscita verso la Porta di I/O B0
----------------------------
  8      RES     3,A         ; Posiziona basso il bit 3
 11      OUT     (2),A       ; Uscita verso la Porta di I/O B0
----------------------------
```

Durata dell'impulso = 19 cicli o 9,5 microsecondi usando un clock di 500 nanosecondi.

Che cosa accade successivamente? La nostra sequenza logica ci può portare al flip-flop FFD, alla destra di FFC o può scendere al monostabile 74121 numerato 36, proprio sotto FFC.

Il monostabile 36 ha i suoi due ingressi A vincolati a terra per cui saranno tutti e due bassi. Se guardate la tabella di funzionamento del 74121 data nel Capitolo 2, troverete che, in questa configurazione, l'uscita di un monostabile è innescata da una transizione basso-alto di B. FFC ($\overline{Q}$) fornisce questo innesco. Ogni altro ingresso B terrà questo monostabile nello stato "off" – che significa che **le uscite Q e $\overline{Q}$ saranno rispettivamente bassa ed alta fino a molto più avanti nel ciclo di stampa, quando FFC commuta nello stato "off";** cioè quando l'uscita $\overline{Q}$ di FFC fa una transizione basso-alto.

Il flip-flop FFD diventa il prossimo dispositivo da simulare.

FLIP-FLOP FFD

Il flip-flop FFD riceve il suo ingresso J direttamente dall'uscita (Q) di FFC; riceve il suo ingresso K dall'uscita ($\overline{Q}$) di FFC. Ricordate che, poichè il monostabile 36 è ancora nello stato "off", la sua uscita $\overline{Q}$ sarà alta; ciò significa che la port AND 12 permetterà semplicemente all'uscita ($\overline{Q}$) di FFC di propagarsi direttamente per diventare l'ingresso (K) di FFD.
Ora il flip-flop riceve gli stessi segnali di reset e di clock di FFC; perciò **il flip-flop FFD andrà nello stato "on" un solo ciclo di clock più tardi di FFC.**

SIMULAZIONE DEL FLIP-FLOP FFD

La simulazione del flip-flop FFD è quasi identica alla simulazione del flip-flop FFC; la differenza principale è che al flip-flop FFD è stato assegnato il bit 3 dell Porta di I/O A1. Di nuovo, ci limiteremo a mettere il flip-flop FFD nello stato "on" e ad assicurarci che il suo posizionamento tra due cicli di stampa sia corretto.

Il flip-flop FFD è messo "off' più avanti nel ciclo di stampa; quindi dobbiamo ricordarci di metterlo "off" più avanti nel programma.

Ecco le modifiche e le aggiunte necessarie del programma:

```
; Esecuzione del programma tra due cicli di stampa
; Posizionare inizialmente ad 1 i bit 1 e 0 della Porta di I/O A1 ed a 0 i bit 3 e 2
         IN    A,(4)        ; La Porta di I/O A1 entra nell'Accumulatore
         OR    3            ; Posiziona i bit 1 e 0
         AND   0F3H         ; Posiziona a 0 i bit 3 e 2
         OUT   (4),A        ; Riporta il risultato
; Test su RETURN STROBE basso
STBHI:   IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
         BIT   4,A          ; Test sul bit RETURN STROBE
         JR    Z,FFB        ; Se è 0, saltare alla simulazione di FFB
         –
         –
         –
; CH RDY basso mette FFA "off". Posiziona ad 1 il bit 0 della Porta di I/O A1
```

```
; Simuliamo pure la commutazione nello stato "on" di FFC. Posizioniamo ad 1 il bit
; 2 della Porta di I/O A1
        IN    A,(4)       ; Carica la Porta di I/O A1 nell'Accumulatore
        OR    5           ; Posiziona i bit 2 e 0 ad 1
        OUT   (4),A       ; Memorizza il risultato
; Mettiamo alto l'impulso START RIBBON MOTION
        IN    A,(2)       ; Ingresso nell'Accumulatore della Porta di I/O B0
        SET   3,A         ; Posiziona alto il bit 3
        OUT   (2),A       ; Uscita verso la Porta di I/O B0            (C)
        RES   3,A         ; Posiziona basso il bit 3
        OUT   (2),A       ; Uscita verso la Porta di I/O B0
; Simuliamo la commutazione nello stato "on" di FFD. Posizioniamo A1 il bit 3
; della Porta di I/Q A1
        IN    A,(4)       ; Ingresso della Porta A1 nell'Accumulatore
        SET   3,A         ; Posiziona ad 1 il bit 3
        OUT   (4),A       ; Memorizza il risultato
```

Se le aggiunte e le modifiche del programma illustrate sopra non sono immediatamente ovvie, confrontiamole con la simulazione del flip-flop C. Non andate avanti se non capite i cambiamenti del programma del flip-flop FFD.

PROGRAMMI RESI PIU' BREVI

Come la simulazione della commutazione nello stato "on" di FFC Ⓐ fu assorbita dalla simulazione di FFB (Ⓑ), così **la simulazione della commutazione nello stato "on" di FFD (Ⓒ) può essere assorbita come segue:**

```
; La sequenza delle istruzioni del nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB
FFB:    IN    A,(4)       ; Carica la Porta di I/O A1 nell'Accumulatore
        RES   1,A         ; Posiziona a 0 il bit 1
        OUT   (4),A       ; Memorizza il risultato
; Simulazione della porta AND 7411 che mette CH RDY basso. Pure la Porta OR
; 7432 mette PW REL basso
        IN    A,(2)       ; Ingresso nell'Accumulatore della Porta di I/O B0
        AND   0FCH        ; Posiziona a 0 i bit 0 ed 1
        OUT   (2),A       ; Memorizza il risultato
; CH RDY basso mette FFA nello stato "off". Posizionare a 1 il bit 0 della Porta di I/O A1
; Simuliamo pure la commutazione nello stato "on" di FFC e FFD. Posizioniamo ad
; 1 i bit 2 e 3 della Porta di I/O A1
        IN    A,(4)       ; Carica la Porta di I/O A1 nell'Accumulatore
        OR    0DH         ; Posiziona ad 1 i bit 3, 2 e 0
        OUT   (4),A       ; Memorizza il risultato
; Mettiamo alto l'impulso START RIBBON MOTION                          (D)
        IN    A,(2)       ; Ingresso nell'Accumulatore della Porta di I/O B0
        SET   3,A         ; Posiziona alto il bit 3
        OUT   (2),A       ; Uscita verso la Porta di I/O B0
        RES   3,A         ; Posiziona basso il bit 3
        OUT   (2),A       ; Uscita verso la Porta di I/O B0
```

Se si combinano le simulazioni (Ⓓ), i flip-flop FFC e FFD commuteranno nello stato "on" esattamente nello stesso istante.

La logica della Figura 3-1 mostra la commutazione nello stato "on" di FFD un solo impulso di clock dopo FFC. Se il periodo del clock è due microsecondi, allora ci sarà un ritardo di due microsecondi tra le commutazioni nello stato "on" dei flip-flop FFD e FFC. Entrambi le nostre simulazioni sono errate.

TEMPORIZZAZIONE E LIMITI DELLA SIMULAZIONE

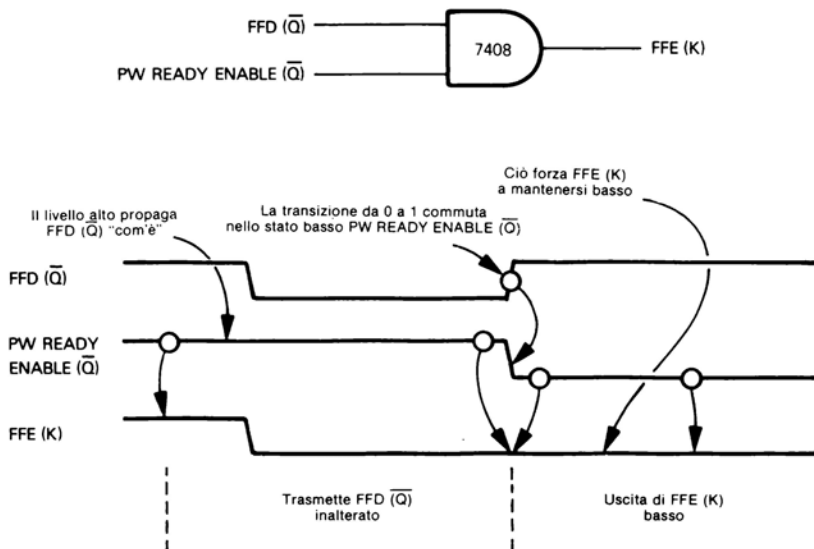
Ciò importa? Onestamente non possiamo dirlo con le informazioni a disposizione. Non conosciamo come la logica esterna usa le uscite di FFC e FFD. **Se l'intervallo di tempo di commutazione tra questi due flip-flop deve essere molto vicino ai due microsecondi, allora la nostra simulazione non funzionerà.** Ognuno dei due flip-flop deve diventare parte di una "logica esterna", o si deve trovare qualche altro metodo per simulare la funzione dell'evento.

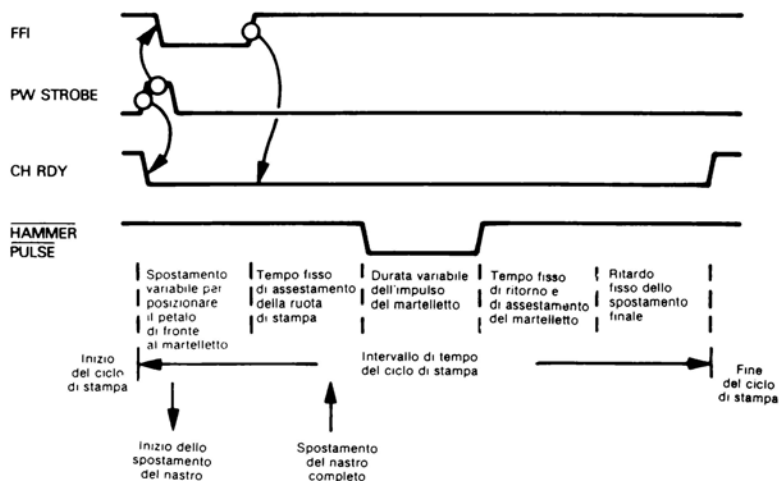
Se la logica esterna richiede qualche ritardo di tempo di commutazione ma non dà importanza alla lunghezza del ritardo di tempo, allora la nostra simulazione del flip-flop FFD (Ⓒ) è adeguata.

E' del tutto possibile che la logica di Figura 3-1 mostri un ritardo di tempo di commutazione tra i flip-flop FFC e FFD solo per definire i fronti di salita e di discesa dell'impulso START RIBBON MOTION; ma abbiamo prestato attenzione a questo impulso alto eseguendo sequenzialmente le istruzioni che mettono 1 poi 0 sul bit 3 della Porta di I/O B0. Finchè si ha a che fare con la logica interna alla Figura 3-1, la necessità di un ritardo di tempo di commutazione tra i flip-flop FFC e FFD scompare. Essendo questo il caso, **supporremo che la logica esterna non abbia bisogno di un ritardo di tempo di commutazione tra i flip-flop FFC e FFD; e noi adotteremo la simulazione combinata più breve identificata da Ⓓ .**

FLIP-FLOP FFE

Il prossimo dispositivo nella nostra sequenza logica è il flip-flop FFE. La circuiteria che circonda questo flip-flop è quasi identica a quella di FFD.

L'ingresso (K) di FFE è collegato all'uscita ($\overline{Q}$) di FFD ed è commutata da un altro componente della porta AND 12. L'altro ingresso di questa porta AND è l'uscita $\overline{Q}$ del monostabile 49. Il monostabile 49 è collegato allo stesso modo del monostabile 36, che abbiamo appena descritto.

La transizione dell'uscita $\overline{Q}$ del flip-flop FFD da 0 a 1 avverrà quando FFD commuta nello stato "off"; questa è la transizione che darà l'avvio al monostabile 49. Perciò, **il monostabile 49 avrà l'uscita $\overline{Q}$ alta finchè il flip-flop FFD non commuterà nello stato "off", che significa che quando FFD commuta nello stato "on" la sua uscita $\overline{Q}$ si propagherà direttamente attraverso la porta AND che lo collega all'ingresso (K) di FFE. (Vedi figura pagina 3-51).**

L'unico aspetto del flip-flop FFE è il modo con cui si genera il suo ingresso J. Questo ingresso è l'AND dell'uscita ($\overline{Q}$) di FFD e il segnale di ingresso FFI. L'uscita Q di FFD andrà alta non appena FFD commuterà nello stato "on"; ma **FFI sarà messo basso dall'inizio del ciclo di stampa fino a che la ruota di stampa non si è posizionata correttamente.** (Abbiamo descritto la funzione di questo segnale d'ingresso precedentemente in questo capitolo). **La temporizzazione associata a FFI può essere illustrata come segue:**

Dal momento che FFI è basso, il flip-flop FFE riceverà un impulso basso sull'ingresso J; ricorderete che gli ingressi J e K bassi mantengono le uscite Q di un flip-flop 74107 nella loro condizione precedente. In tale modo, **si usa il segnale d'ingresso FFI per creare il primo ritardo di tempo del ciclo di stampa; un ritardo di tempo variabile necessario per spostare il petalo desiderato della ruota di stampa di fronte al martelletto di stampa. La simulazione di questo ritardo di tempo è abbastanza semplice; essa può essere illustrata come segue:**

```
; Mettiamo alto l'impulso START RIBBON MOTION
        IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
        SET   3,A          ; Posiziona il bit 3 alto
        OUT   (2),A        ; Uscita verso la Porta di I/O B0
        RES   3,A          ; Posiziona basso il bit 3
        OUT   (2),A        ; Uscita verso la Porta di I/O B0
```

```
; Test sull'ingresso di decodifica della velocità per creare un ritardo nel movimento
; della ruota di stampa
VLDC:   IN    A,(0)        ; La Porta di I/O A0 entra nell'Accumulatore
        RLA                ; Sposta il bit 7 nel Carry
        JR    NC,VLDC      ; Rimane in loop se il Carry è zero
; Alla fine del ritardo simulate la commutazione nello stato "on" di FFE
        IN    A,(4)        ; La Porta di I/O A1 entra nell'Accumulatore
        RES   5,A          ; Posiziona a 0 il bit 5
        SET   4,A          ; Posiziona ad 1 il bit 4
        OUT   (4),A        ; Uscita del risultato
```

RITARDO DI TEMPO DI LUNGHEZZA VARIABILE

SALTO SE NON C'E' CARRY

Per generare il ritardo di tempo iniziale, si esegue semplicemente un loop continuo di programma che fa entrare il contenuto della Porta di I/O A0 nell'Accumulatore. Il bit 7 della Porta di I/O A0 è stato assegnato al segnale di ingresso FFI. Saggiamo questo bit spostandolo nello stato del Carry. Se lo stato del Carry ha quindi un contenuto 0, FFI dovrà essere ancora basso; così si rimane nel loop. Non appena si sposta un 1 nello stato del Carry, l'istruzione JR NC creerà un risultato "falso"; si esegue la successiva sequenza di istruzioni e ci troviamo fuori del loop del ritardo di tempo:

Saltare se non c'è Carry, significa saltare se il Carry è 0. "Saltare" significa non andare alla successiva istruzione in sequenza, ma andare a VLDC.

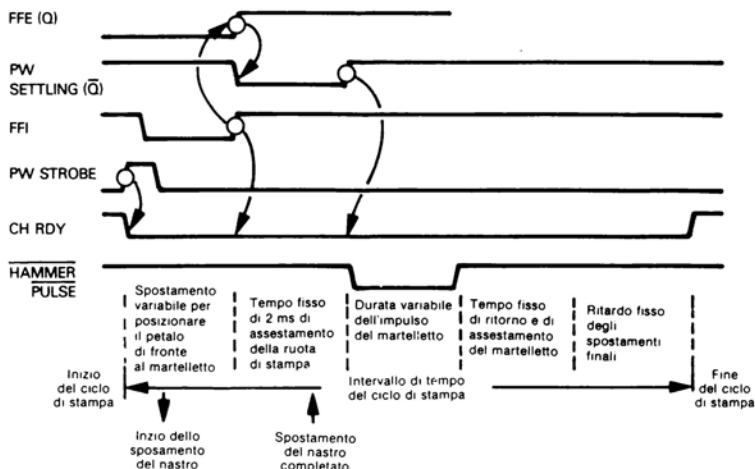
Le ultime quattro istruzioni della simulazione di FFE mostrano entrambi le uscite di questo flip-flop mentre diventano segnali di uscita. Ciò soddisfa i requisiti della Figura 3-1. Posizioniamo perciò a 0 il bit 5 (che rappresenta l'uscita $\overline{Q}$) e posizioniamo ad 1 il bit 4 (che rappresenta l'uscita Q).

Si dovrà modificare la sequenza di istruzioni eseguita tra due cicli di stampa per assicurarci che il bit 5 sia inizialmente posizionato ad 1, mentre il bit 4 sia posizionato inizialmente a 0. Ecco le modifiche richieste:

```
; Esecuzione del programma tra due cicli di stampa
; Posizioniamo inizialmente ad 1 i bit 5, 1 e 0 della Porta di I/O A1, a 0 i bit 4,
; 3 e 2
        IN    A,(4)        ; La Porta di I/O A1 entra nell'Accumulatore
        OR    23H          ; Posiziona ad 1 i bit 5, 1 e 0
        AND   0E3H         ; Posiziona a 0 i bit 4, 3 e 2
        OUT   (4),A        ; Ritorno del risultato
; Test su RETURN STROBE basso
STBHI:  IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
        BIT   4,A          ; Test sul bit RETURN STROBE
        JR    Z,FFB        ; Se è 0, saltare alla simulazione di FFB
```

IL MONOSTABILE PW SETTLING

Il monostabile PW SETTLING è il dispositivo 74121 di coordinate B6 della Figura 3-1. Abbiamo descritto questo dispositivo nel Capitolo 2. Con i suoi due ingressi A collegati a terra, **questo monostabile è innescato da una transizione basso-alto sul suo ingresso B. Poichè l'ingresso B è attaccato all'uscita Q di FFE, questa transizione avviene non appena il flip-flop FFE commuta nello stato "on".**

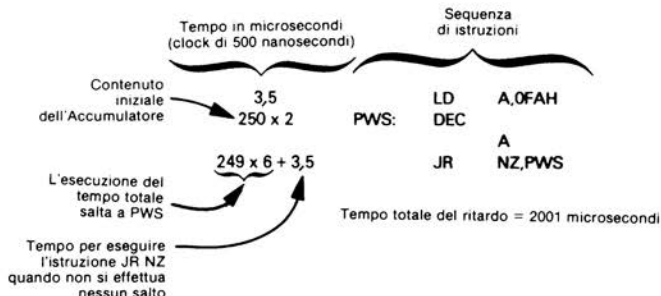
Il monostabile PW SETTLING ha un ritardo di tempo di due millisecondi. Questo ritardo è dato dalla combinazione esterna del condensatore e della resistenza indicate con C1 ed R1. Perciò, non appena FFE va "on", il monostabile PW SETTLING mette un impulso basso sull'uscita $\overline{Q}$ per due millisecondi:

SIMULAZIONE DEL MONOSTABILE PW SETTLING

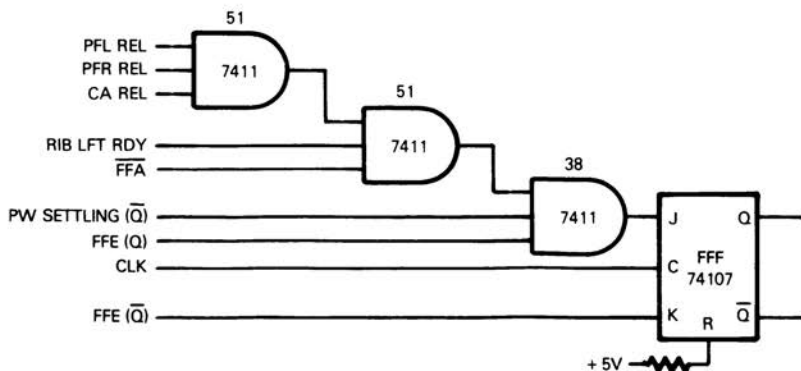
SIMULAZIONE DEL RITARDO DI TEMPO DI UN MONOSTABILE

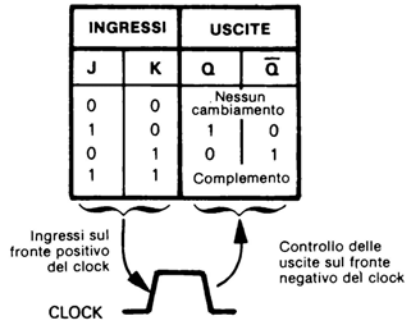
La simulazione del ritardo di tempo di un monostabile è abbastanza semplice e può essere illustrata come segue:

```
; Mettiamo alto l'impulso START RIBBON MOTION
        IN    A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        SET   3,A        ; Posiziona il bit 3 alto
        OUT   (2),A      ; Uscita verso la Porta di I/O B0
        RES   3,A        ; Posiziona basso il bit 3
        OUT   (2),A      ; Uscita verso la Porta di I/O B0
; Test sull'ingresso di decodifica della velocità per creare un ritardo del movimento
; della ruota di stampa
VLDC:   IN    A,(0)      ; La Porta di I/O A0 entra nell'Accumulatore
        RLA              ; Sposta il bit 7 nel Carry
        JR    NC,VLDC    ; Rimani in loop se il Carry è zero
; Alla fine del ritardo si simula la commutazione nello stato "on" di FFE
        IN    A,(4)      ; La Porta di I/O A1 entra nell'Accumulatore
        RES   5,A        ; Posiziona a 0 il bit 5
        SET   4,A        ; Posiziona ad 1 il bit 4
        OUT   (4),A      ; Uscita del risultato
```

```
; Simulazione del ritardo di tempo di 2 ms di PW SETTLING
        LD   A,0FAH      ; Carica la costante del ritardo del tempo iniziale
PWS:    DEC  A           ; Decrementa l'Accumulatore
        JR   NZ,PWS      ; Ridecrementa se il risultato è diverso da zero
```

Ci sono due istruzioni nel loop del ritardo di tempo: DEC A e JR NZ; in tale modo, il ritardo di tempo può essere calcolato come segue:

FLIP-FLOP FFF

Una volta che il monostabile PW SETTLING ha realizzato il suo tempo, siamo pronti per alimentare il martelletto di stampa. Il multivibratore 555 sta realmente generando l'impulso che fa alimentare il martelletto di stampa, ma è molto più importante assicurarsi che il martelletto di stampa non sia alimentato mentre una parte qualunque dei meccanismi di stampa o del carrello si sta muovendo. Il monostabile 555 è perciò innescato dal flip-flop FFF, che, a sua volta, è messo "on" da un ingresso J che è l'AND di molti segnali di protezione. Isoliamo il flip-flop FFF ed esaminiamo i suoi ingressi.

Con il suo ingresso Clear (R) collegato a +5V, il flip-flop FFF ha la seguente tabella di funzionamento.

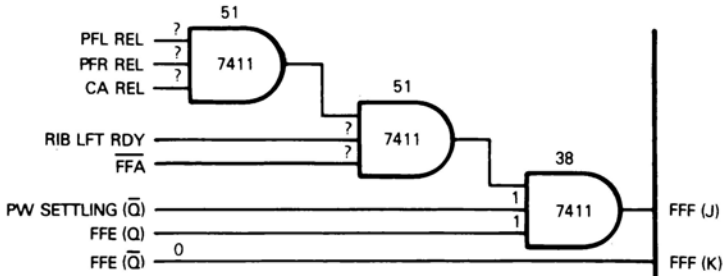
INGRESSI		USCITE	
J	K	Q	$\overline{Q}$
0	0	Nessun cambiamento	
1	0	1	0
0	1	0	1
1	1	Complemento	

Tra due cicli di stampa FFE è "off", così l'ingresso K di FFF è alto. L'ingresso J sarà basso, poichè l'uscita (Q) di FFE sarà bassa, ed essa partecipa a creare il J di FFF.

Perciò tra due cicli di stampa, il flip-flop FFF è "off", poichè un ingresso J basso ed un ingresso K alto generano uscite stabili con Q=0, $\overline{Q}$=1; questa è una caratteristica di un flip-flop nella condizione "off".

Quando FFE commuta nello stato "on", esso mette un impulso basso sull'ingresso K di FFF. Dal momento che l'ingresso J è pure basso, non avviene nessun cambiamento. Non appena i sette segnali che contribuiscono al J di FFF sono alti, il flip-flop FFF riceverà un ingresso J alto; ciò metterà "on" il flip-flop FFF — l'uscita Q è alta e l'uscita $\overline{Q}$ è bassa.

SIMULAZIONE DEL FLIP-FLOP FFF

Uscendo dalla simulazione del flip-flop FFE, sappiamo che le uscite Q e $\overline{Q}$ di FFE hanno i livelli giusti per mettere FFF "on".

Provenendo dalla simulazione del monostabile PW SETTLING, l'uscita $\overline{Q}$ deve essere alta:

Tutto ciò che bisogna fare è di verificare i rimanenti cinque segnali di sincronizzazione; non appena essi sono tutti alti, simuleremo la commutazione nello stato "on" del flip-flop FFF. La sequenza delle istruzioni è la seguente:

```
; Test sull'ingresso di decodifica della velocità per creare un ritardo del movimento
; della ruota di stampa
VLDC:   IN    A,(0)        ; La Porta di I/O A0 entra nell'Accumulatore
        RLA                ; Sposta il bit 7 nel Carry
        JR    NC,VLDC      ; Rimani in loop se il Carry è zero
; Alla fine del ritardo si simula la commutazione nello stato "on" di FFE
        IN    A,(4)        ; La Porta di I/O A1 entra nell'Accumulatore
        RES   5,A          ; Posiziona a 0 il bit 5
        SET   4,A          ; Posiziona ad 1 il bit 4
        OUT   (4),A        ; Uscita del risultato
; Simulazione del ritardo di tempo di 2 ms di PW SETTLING
        LD    A,0FAH       ; Carica la costante del ritardo di tempo iniziale
PWS:    DEC   A            ; Decrementa l'Accumulatore
        JR    NZ,PWS       ; Ridecrementa se il risultato è diverso da zero
; Simulazione della commutazione nello stato "on" del flip-flop FFF
FFF:    IN    A,(0)        ; Il contenuto della Porta di I/O A0 entra nell'Ac-
                           ; cumulatore
        CPL                ; Complementa per saggiare i bit ad 1
        AND   1FH          ; Isola i bit da 0 a 4
        JR    NZ,FFF       ; Se qualche bit è 1, rimanere in loop
        IN    A,(4)        ; Posiziona ad 1 il bit 6 della Porta di I/O A1
        SET   6,B
        OUT   (4),A
```

Per ora, voi potete capire le istruzioni così come esse vengono aggiunte nel programma.

Le prime quattro istruzioni caricano semplicemente il contenuto della Porta di I/O A0 nell'Accumulatore e verificano se c'è un 1 nei cinque bit di peso minore. Finchè tutti i cinque bit non saranno 1, il programma rimarrà nel loop di quattro istruzioni che comincia con IN A,(0) e finisce con JR NZ,FFF.

Quando i bit da 0 a 4 sono tutti uguali ad 1, l'istruzione CPL cambia tutti questi bit a 0:

			Contenuto dell'accumulatore	
FFF:	IN	A,(0)	X X X 1 1 1 1 1	
	CPL		$\overline{X}\overline{X}\overline{X}$ 0 0 0 0 0	
	AND	1FH	0 0 0 1 1 1 1 1	
			0 0 0 0 0 0 0 0	Stato Zero = 1
	JR	NZ,FFF	Ritorna a FF solo se lo stato Zero = 0	
	IN	A,(4)	Contiua qui se lo stato Zero = 1	

L'istruzione JR NZ non riporta più l'esecuzione del programma a FFF, ma permette l'esecuzione della prossima istruzione in sequenza.

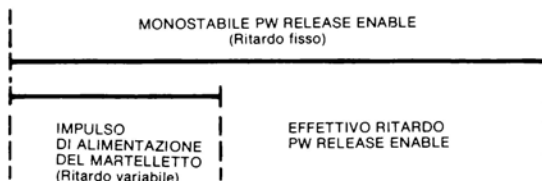
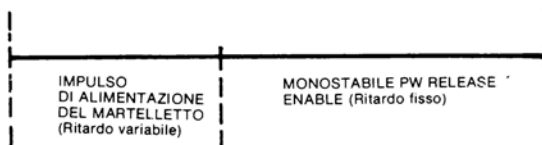
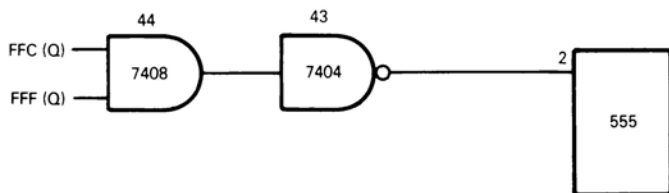
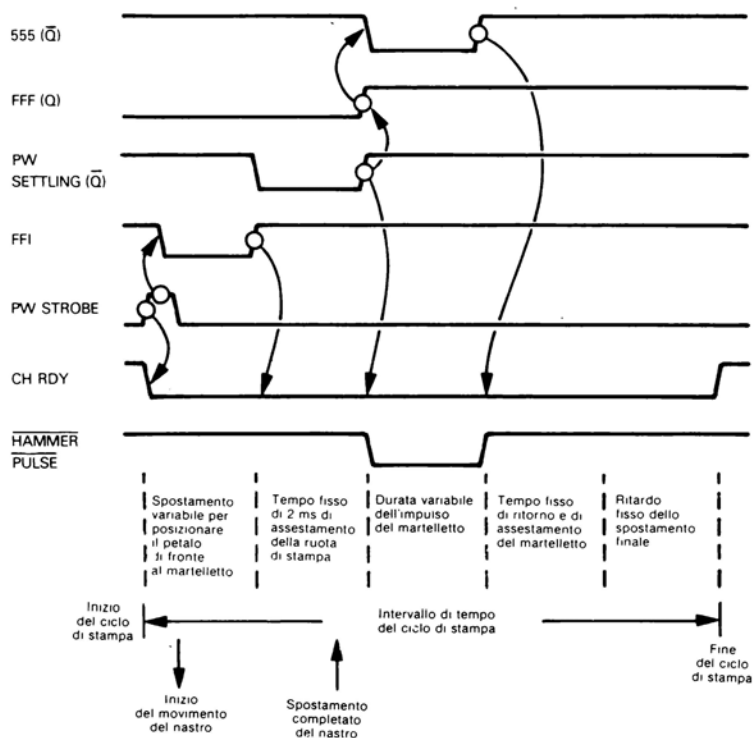
Possiamo fare la modifica finale alla sequenza di istruzioni che posiziona correttamente lo stato del flip-flop tra due cicli di stampa. Ecco come si finisce:

```
; Esecuzione del programma tra due cicli di stampa
; Posizionamento iniziale ad 1 dei bit 5, 1 e 0 della Porta di I/O A1, a 0 dei bit 6,
; 3 e 2.
        IN    A,(4)        ; La Porta di I/O A1 entra nell'Accumulatore
        OR    23H          ; Posizionamento dei bit 5, 1 e 0 ad 1
        AND   0A3H         ; Posizionamento a 0 dei bit 6, 4, 3 e 2
        OUT   (4),A        ; Ritorno del risultato
```

Che cosa succede quando il flip-flop FFF commuta nello stato "on"?

L'uscita (Q) di FFF va al pin 9 dell'AND 37 di coordinate C6. Ciò fa parte della logica che contribuisce al segnale PW REL. Tuttavia **la transizione dell'uscita (Q) di FFF da basso ad alto non è significativa,** poichè l'altro ingresso della porta AND 37 è l'uscita ($\overline{Q}$) di FFD che è correntemente bassa. L'uscita (Q) di FF è collegata alla porta AND 37 per tenere basso PW REL precedentemente nel ciclo di stampa, quando ($\overline{Q}$) di FFD è alta.

Le uscite Q e $\overline{Q}$ di FFF contribuiscono agli ingressi J e K di FFC. ($\overline{Q}$) di FFF è un ingresso della porta AND 12, la cui uscita diventa l'ingresso (J) di FFC. L'altro ingresso di questa porta AND è l'uscita della porta AND 37 di coordinate A4, che è costantemente alta da quest'istante nel ciclo di stampa; quindi quando l'uscita ($\overline{Q}$) di FFF va bassa va pure basso l'ingresso (J) di FFC. L'ingresso K di FFC è l'uscita (Q) di FFF. **FFC sarà perciò commutato nello stato "off" quando K va alto, e ciò non accadrà finchè FFF non commuterà nello stato "on".**

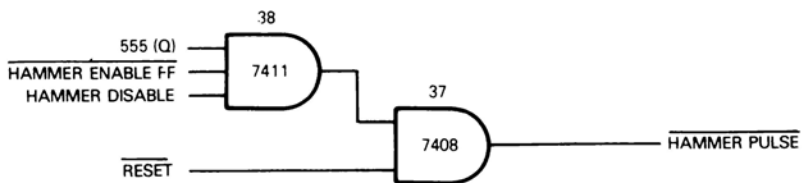
Nella nostra simulazione, tuttavia, posporremo la commutazione nello stato "off" di FFC fino alla fine di $\overline{\text{HAMMER PULSE}}$. Ciò avviene perchè lo scopo della commutazione nello stato "off" di FFC è di innescare il monostabile PW RELEASE ENABLE, che crea il ritardo di tempo necessario al martelletto di stampa per tornare indietro. Così, invece di usare ritardi paralleli:

implementeremo ritardi seriali, che si accordano con più immediatezza con la necessità della logica:

L'impulso che alimenta il martelletto è generato dal multivibratore 555. Perciò il multivibratore 555 fornisce il prossimo evento nella nostra sequenza cronologica; esso è fatto partire da una transizione alto-basso sul pin 2. Questo ingresso sul pin è creato come segue:

Ecco la sequenza di eventi che deve essere simulata:

IL MULTIVIBRATORE 555

Confrontate il modo di collegamenti del multivibratore 555 di Figura 3-1 con la descrizione del multivibratore data nel Capitolo 2; vedrete che il **flip-flop FFB mette il multivibratore nello stato "off" tra due cicli di stampa** posizionando un impulso basso sul pin 4. **L'uscita (Q) di FFF innesca il multivibratore,** come abbiamo appena descritto.

IMPULSO VARIABILE DI MONOSTABILE

La durata dell'impulso di uscita del monostabile è controllata dagli ingressi H1 ÷ H6. Uno di questi ingressi sarà vero mentre gli altri cinque saranno falsi; in tale modo il multivibratore, una volta innescato, farà uscire un impulso "alto" che avrà sei durate possibili.

L'uscita del multivibratore monostabile 555 sarà eventualmente invertita per dare la uscita $\overline{\text{HAMMER PULSE}}$; tuttavia, perchè ci sia l'uscita HAMMER PULSE, devono

essere alti anche gli altri ingressi delle porte AND 37 e 38, di coordinate C7. Possiamo rappresentare la logica di HAMMER PULSE come segue:

Dovremo semplicemente fare un test sull'ingresso HAMMER ENABLE FF prima di generare un'uscita HAMMER PULSE.

Si deve simulare l'interruttore HAMMER DISABLE.

Possiamo ignorare RESET, poichè la logica di azzeramento è stata simulata tra due cicli di stampa.

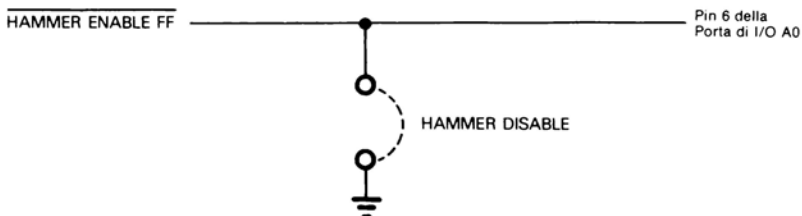
SIMULAZIONE DEL MULTIVBRATORE 555

La simulazione del multivibratore 555 consiste nella seguente sequenza logica:

1) **Determinare se sono state soddisfatte le condizioni affinchè l'uscita del monostabile 555 sia trasmessa come uscita di HAMMER PULSE.**
2) **Esaminare gli ingressi da H1 a H6. Basandosi su questi ingressi, creare uno di sei possibili ritardi.**
3) **Se sono state soddisfatte le condizioni per un'uscita di HAMMER PULSE, trasformare l'uscita del monostabile 555 in un'uscita HAMMER PULSE.**

Guardiamo dapprima la logica di abilitazione dell'uscita di HAMMER PULSE. Il test sulla condizione HAMMER ENABLE FF è abbastanza semplice; essa è stata assegnata al pin 6 della Porta di I/O A0.

LOGICA ESCLUSA DAL MICROCALCOLATORE

Ma non esistono interruttori in programmi in linguaggi assembly; come simuleremo la disabilitazione del martelletto (HAMMER DISABLE)? Potremmo assegnare il pin rimanente – il pin 5 della Porta di I/O A0 – a un seganle d'ingresso generato da un interruttore esterno. Sarebbe semplice inserire questo interruttore sul percorso di HAMMER ENABLE FF come segue:

Ignoreremo quindi l'interruttore di disabilitazione del martelletto ed abiliteremo una uscita HAMMER PULSE, mettendo alto l'ingresso HAMMER ENABLE FF.

Che cosa si può dire sulle sei possibili durate dell'uscita del multivibratore 555? Abbiamo descritto nel Capitolo 2 come si può creare un ritardo di tempo caricando un valore a 16 bit in una coppia di registri, decrementando poi questa coppia di registri in un loop del programma, rimanendo nella loop di programma finchè non si ottiene un decremento a zero. **Scegliere uno dei sei possibili ritardi di tempo è semplice come scegliere una delle sei possibili costanti di tempo iniziali. Possiamo ora simulare il multivibratore 555 come segue:**

```
FFF:    IN    A,(0)      ; Il contenuto della Porta di I/O A0 entra nell'Ac-
                         ; cumulatore
        CPL              ; Complemento per test sui bit ad 1
        AND   1FH        ; Isola i bit da 0 a 4
        JR    NZ,FFF     ; Se qualche bit è ad 1, rimanere in loop
        IN    A,(4)      ; Posiziona ad 1 il bit 6 della Porta di I/O A1
        SET   6,B
        OUT   (4),A
; Test su HAMMER ENABLE FF
        IN    A,(0)      ; La Porta di I/O A0 entra nell'Accumulatore
        BIT   6,A        ; Test sul bit 6
        JR    Z,HPO      ; Se zero, superare il posizionamento a basso di
                         ; HAMMER PULSE
; HAMMER ENABLE FF è alto, così HAMMER PULSE deve essere basso in uscita
; quindi posizionare a 0 il bit 2 della Porta di I/O B0
        IN    A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        RES   2,(A)      ; Posiziona a 0 il bit 2
        OUT   (2),A      ; Uscita del risultato
; Calcolo del ritardo di tempo
HPO:    LD    HL,DELY    ; Carica l'indirizzo di base del ritardo nella coppia HL
        IN    A,6        ; Il selettore (Porta B1) entra nell'Accumulatore
HP1:    RRA              ; Rotazione a destra dell'Accumulatore con Carry
        INC   HL         ; Incremento di 2 del contenuto di HL
        INC   HL
        JR    NC,HP1     ; Se non c'è Carry, ruotare ed incrementare di nuovo
        LD    E,(HL)     ; Carica una costante di tempo di ritardo a 16 bit in DE
        INC   HL
        LD    (HL)
TDLY:   DEC   DE         ; Esecuzione del loop del ritardo di tempo
        LD    A,D
        OR    E
        JR    NZ,TDLY
; Uscita ancora alta su HAMMER PULSE
        IN    A,(2)      ; La Porta di I/O B0 entra nell'Accumulatore
        SET   2,A        ; Posiziona ad 1 il bit 2
        OUT   (2),A      ; Uscita del risultato
```

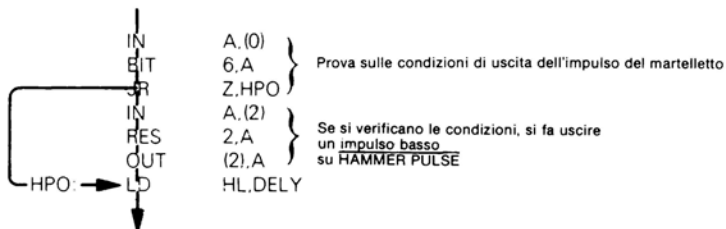
Paragonato agli altri dispositivi che abbiamo già simulato, il multivibratore 555 richiede una quantità maggiore di istruzioni per la simulazione. Sebbene possa sembrare che ci sia molto da capire, in realtà, la logica è molto semplice; così consideriamone un pezzo per volta.

ABILITAZIONE DEI SEGNALI

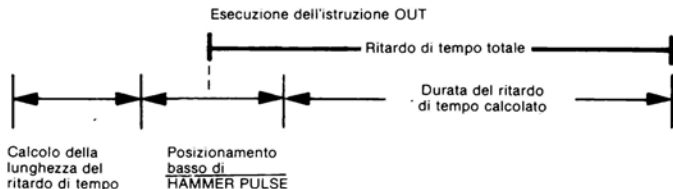
Inizialmente saggiamo HAMMER ENABLE FF. HAMMER PULSE sarà messo basso in uscita se HAMMER ENABLE FF è alto. Le tre istruzioni che verificano lo stato di HAMMER

ENABLE FF sono:

```
    IN    A,(0)       ; La Porta di I/O A0 entra nell'Accumulatore
    BIT   6,A         ; Test sul bit 6
    JR    Z,HPO       ; Se zero, superare il posizionamento a basso di
                      ; HAMMER PULSE
```

Ci sono due aspetti di queste tre istruzioni che bisogna spiegare. Dapprima c'è da implementare della logica. Stiamo determinando se si sono verificate le condizioni per fare uscire basso HAMMER PULSE. Se le condizioni si sono verificate, l'istruzione JR Z,HPO si dirama lungo la sequenza di istruzioni che mettono basse HAMMER PULSE.

SEQUENZA DEGLI EVENTI

Facciamo uscire basso HAMMER PULSE prima di calcolare la durata dell'impulso di tempo; perchè ciò? Il motivo è di risparmiare tempo. Le istruzioni che calcolano la lunghezza del ritardo di tempo si possono eseguire all'inizio del ritardo di tempo.

Potremmo avere appena calcolato facilmente il ritardo di tempo, quindi messo basso HAMMER PULSE e infine avere eseguito il ritardo di tempo; gli eventi sarebbero avvenuti cronologicamente come segue:

Eventi che si sovrappongono nel tempo hanno una sensibilità molto maggiore.

Il metodo reale usato per calcolare il ritardo di tempo abbisogna di una piccola spiegazione. **Alla fine del nostro programma ci saranno 12 byte di memoria nei quali sono memorizzate sei costanti a 16 bit.** Ecco come apparirà il programma sorgente:

```
; Nuova uscita di un alto su HAMMER PULSE
        IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
        SET   2,A          ; Posiziona ad 1 il bit 2
        OUT   (2),A        ; Uscita del risultato
        -
        -
        -
        ORG   DELY+2
        DEFW  PPQQH        ; Ritardo di tempo H1
        DEFW  RRSSH        ; Ritardo di tempo H2
        DEFW  TTUUH        ; Ritardo di tempo H3
        DEFW  VVWWH        ; Ritardo di tempo H4
        DEFW  XXYYH        ; Ritardo di tempo H5
        DEFW  ZZOOH        ; Ritardo di tempo H6
```

Per rappresentare valori esadecimali si sono usate le lettere da 0 a Z. I sei ritardi di tempo possono essere rappresentati da un qualsiasi valore numerico, variabile da 0000_{16} a $FFFF_{16}$.

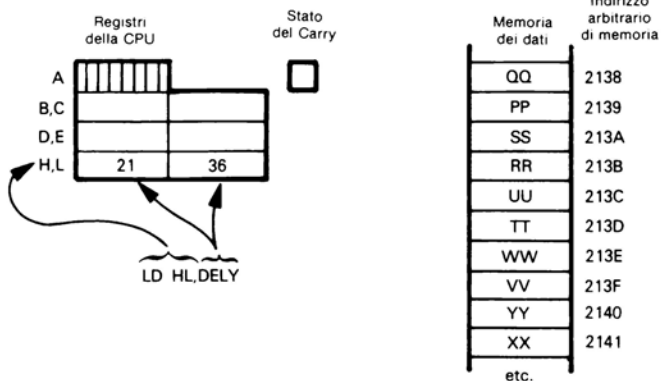
L'indirizzo del primo byte di memoria in cui è memorizzato il primo ritardo di tempo è dato dall'espressione DELY+2. Supponiamo che questa locazione di memoria sia 2138:

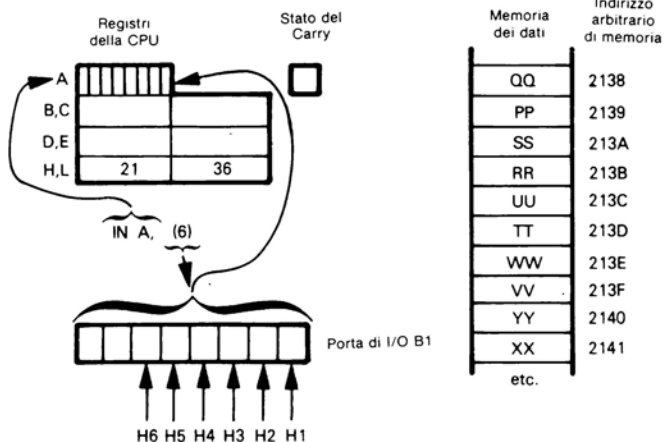
Memoria dati	Indirizzo arbitrario di memoria
QQ	2138
PP	2139
SS	213A
RR	213B
UU	213C
TT	213D
WW	213E
VV	213F
YY	2140
XX	2141
OO	2142
ZZ	2143

Ogni valore a 16 bit occuperà 2 locazioni di memoria. L'assembler dello Z80 metterà il byte meno significativo nella locazione di indirizzo minore. Ciò è congruente con la rappresentazione del codice oggetto degli indirizzi e dei valori di dati immediati a 16 bit, come abbiamo ricordato nel Capitolo 2.

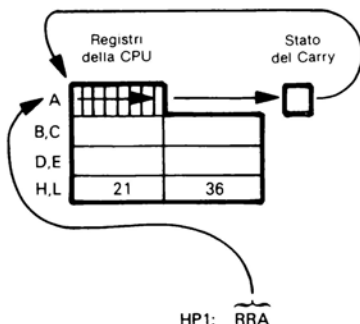
DELY è una label alla quale deve essere assegnato il valore 2136. L'assegnazione è fatta usando una direttiva Equate che apparirà all'inizio del programma, come segue:

```
DELY     EQU     2136H
```

Cominciamo ora il calcolo del ritardo di tempo caricando l'indirizzo DELY nella coppia di registri HL. Supponiamo che la label DELY abbia il valore 2136, come illustrato sopra. Dopo l'esecuzione dell'istruzione LD HL,DELY, ecco la situazione:

La istruzione successiva, IN A,(6), carica nell'Accumulatore il contenuto della Porta di I/O B1. Ricordiamo, dalla nostra discussione sui segnali d'ingresso, che, dei sei segnali d'ingresso H1 ÷ H6, solo uno sarà alto mentre gli altri cinque saranno bassi.

Perciò, **dopo l'esecuzione dell'istruzione IN, l'Accumulatore conterrà un solo 1 in uno dei sei bit di peso minore:**

CALCOLO DELL'INDIRIZZO DELLA MEMORIA DATI

Possiamo calcolare l'indirizzo del ritardo di tempo richiesto sommando 2 al contenuto della coppia di registri HL un numero di volte dato dalla posizione del bit 1 nello Accumulatore. Ciò può essere illustrato come segue:

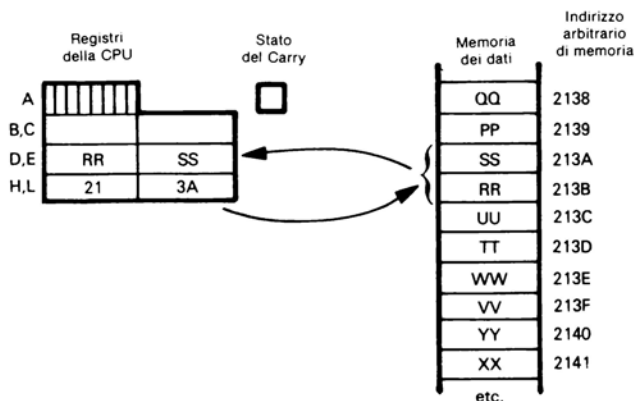
① Rotazione del contenuto dell'Accumulatore a destra di un bit con Carry:

Memoria dei dati	Indirizzo arbitrario di memoria
QQ	2138
PP	2139
SS	213A
RR	213B
UU	213C
TT	213D
WW	213E
VV	213F
YY	2140
XX	2141
etc.	

② Somma di 2 alla coppia di registri HL:

③ Se lo stato di Carry non è 1, tornare a ① ; altrimenti, HL contiene l'indirizzo corretto.

La logica per fare la somma dell'indirizzo richiesta è fornita da queste quattro istruzioni:

```
HP1:    RRA             ; Rotazione a destra dell'Accumulatore con Carry
        INC   HL        ; Incremento di due del contenuto di HL
        INC   HL
        JR    NC,HP1    ; Se non c'è Carry, ruotare ed incrementare di nuovo
```

Quando l'istruzione JR NC fa sì che l'esecuzione del programma continui con la successiva istruzione in sequenza piuttosto che tornare a HP1, HL conterrà l'indirizzo del primo byte della costante di tempo iniziale del ritardo. **Ora che i registri H e L indirizzano il corretto ritardo di tempo, carichiamo l'appropriata costante del ritardo a 16 bit in D ed E.** Supponiamo che H2 sia il segnale d'ingresso alto; questo è il risultato:

La costante di ritardo scelta RRSS è messa nei registri D ed E da queste tre istruzioni:

```
        LD    E,(HL)     ; Sposta il contenuto del byte 213A nel registro E
        INC   HL         ; Indirizza il byte 213B
        LD    D,(HL)     ; Sposta il contenuto del byte 213B nel registro D
```

Notate che per primo carichiamo il Registro di peso minore E, poichè il byte di peso minore è l'indirizzo minore.

Il ritardo in tempo reale è creato da questo loop di istruzioni, che è stato descritto nel Capitolo 2:

```
TDLY:   DEC   DE         ; Decremento del contatore del ritardo
        LD    A,D        ; Prova dello 0 in DE facendo l'OR di D ed E con
                         ; l'Accumulatore
        OR    E
        JR    NZ,TDLY    ; Ritorno se il risultato è diverso da zero
```

Le ultime tre istruzioni mettono alto $\overline{\text{HAMMER PULSE}}$, senza fare nessun test caso mai $\overline{\text{HAMMER PULSE}}$ fosse basso. Questa logica funzionerà poichè facendo uscire alto $\overline{\text{HAMMER PULSE}}$, se esso fosse già alto, non avrebbe alcun effetto discernibile. In queste circostanze, il tempo richiesto per eseguire le ultime tre istruzioni è semplicemente perso. Poichè ci vorrebbero tre istruzioni per vedere se $\overline{\text{HAMMER PULSE}}$ è basso, la perdita di tempo è giustificata.

CALCOLO DEL RITARDO DI TEMPO

Pensiamo ora un poco al tempo che ci vorrà per calcolare il ritardo di tempo. I tempi di esecuzione delle istruzioni pertinenti sono elencate come segue:

```
Cicli          Istruzione
                 IN     A,(2)
                 RES    2,A
                 OUT    (2),A      <---- Qui inizia il basso su
 10     HP0:     LD     HL,DELY          HAMMER PULSE
 10              IN     A,(6)
  4     HP1:     RRA              }
  6              INC    HL        }  Queste quattro istruzioni saranno ese-
  6              INC    HL        }  guite da 1 a 6 volte. Nel loop ci sono 28
7/12             JR     NC,HP1    }  cicli.
  7              LD     E,(HL)
  6              INC    HL
  7              LD     D,(HL)
  6     TDLY:    DEC    DE        }
  4              LD     A,D       }  Queste quattro istruzioni costituisco-
  4              OR     E         }  no il ritardo di tempo. In questo loop
7/12             JR     NZ,TDLY   }  ci sono 26 cicli.
 10              IN     A,(2)
  8              SET    2,A
 11              OUT    (2),A      <---- Qui finisce l'impulso basso su
----                                     HAMMER PULSE
113
```

Supponendo un clock di 500 nanosecondi, il tempo richiesto per iniziare e terminare il segnale $\overline{\text{HAMMER PULSE}}$ è dato da:

$$56{,}5 - 13 - 14 + 14N \text{ microsecondi}$$

dove N è un numero compreso tra 1 e 6, rappresentante la posizione nell'Accumulatore del bit che deve essere posizionato ad 1. **In tale modo il tempo d'inizio e di termine varierà tra 43,5 microsecondi e 113,5 microsecondi.** Il tempo più breve si applica per N = 1 (H1), mentre il più lungo per N = 6 (H6).

Questi tempi devono essere sottratti dai ritardi generati successivamente. Per esempio, supponiamo che H1 alto richieda al 555 di far uscire un segnale di monostabile che sia alto per 1,65 millisecondi (approssimativamente); per cui sarà sufficiente un ritardo di 1,6 millisecondi sommato col tempo di posizionamento di 43,5 microsecondi.

IL FLIP-FLOP PW RELEASE ENABLE

Non appena l'uscita del monostabile 555 diventa nuovamente alta, si è simulata la commutazione nello stato "on" del flip-flop FFC. Quando FFC va "off", la sua uscita $\overline{Q}$ fa una transizione basso-alto e ciò innesca il monostabile PW RELEASE ENABLE. Questo è un monostabile 74121 identificato dal 36 in coordinate E2. Lo scopo di questo monostabile è di dare al martelletto di stampa il tempo di riposizionarsi prima che si faccia un tentativo di riposizionare la ruota di stampa. **Ciò è stato illustrato come ritardo di tempo di posizionamento e di ritorno del martelletto.**

SIMULAZIONE DEL FLIP-FLOP PW RELEASE ENABLE

RITARDO DI TEMPO

In realtà questa è una simulazione composta da due parti; dapprima dobbiamo simulare la commutazione nello stato "off" del flip-flop FFC, quindi eseguire il ritardo di tempo appropriato. E' sufficiente un ritardo di tempo di tre millisecondi. Le istruzioni che mettono "off" il flip-flop FFC saranno eseguite nel ritardo di tempo di tre millisecondi.

Il ritardo di tempo calcolato sarà perciò un po' minore di tre millisecondi. Ecco la sequenza di istruzioni appropriata:

```
; Uscita ancora alta su HAMMER PULSE
        IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        SET   2,A         ; Posiziona ad 1 il bit 2
        OUT   (2),A       ; Uscita del risultato
; Commutazione nello stato "off" del flip-flop FFC
        IN    A,(4)       ; Posiziona a 0 il bit 2 della Porta di I/O A1
        RES   2,A
        OUT   (4),A
; Esecuzione di un ritardo di tempo di 3 millisecondi
        LD    DE,230      ; Carica la costante di tempo in D, E
PWRL:   DEC   DE          ; Decremento della coppia di registri
        LD    A,D         ; Test per lo zero
        OR    E
        JR    NZ,PWRL     ; Ridecrementa se non è zero
```

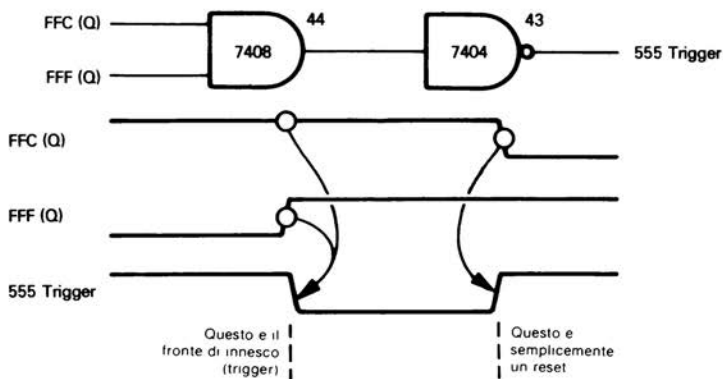
Notate che la costante di tempo iniziale è stata identificata come un numero decimale, 230. La costante di tempo potrebbe essere specificata da un numero esadecimale in questo modo:

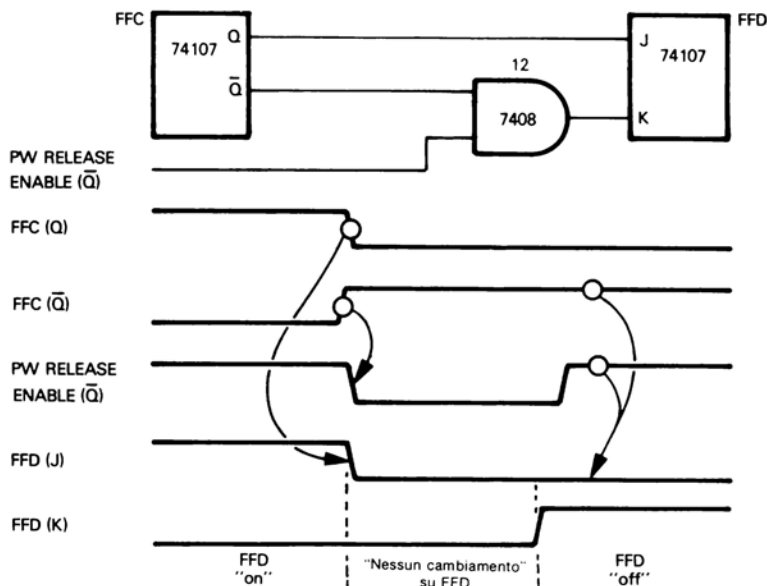
```
        LD    DE,0E6H
```

Supponendo un clock di 500 nanosecondi, le tre istruzioni che precedono il loop del ritardo di tempo (RES OUT e LD) saranno eseguite in 14,5 microsecondi, e le quattro istruzioni nel loop del ritardo di tempo saranno eseguite in 13 microsecondi. Perciò il tempo totale del ritardo è dato dall'equazione:

$$229 \times 13 + 10{,}5 + 14{,}5 = 3002 \text{ microsecondi}$$

Onestamente il ritardo di tempo di tre millisecondi non è un numero critico; 2,5 o 3,5 millisecondi andrebbero probabilmente bene lo stesso, così la nostra preoccupazione sui 10 microsecondi in questa ipotesi non è significativa. Tuttavia, nella vostra prossima applicazione la durata del ritardo di tempo può essere molto critica; allora le considerazioni sulla temporizzazione discussa sopra saranno molto significative.

Per determinare che cosa succede alla conclusione del ritardo di tempo PW RELEASE, dobbiamo guardare le uscite $\overline{Q}$ e Q di FFC. L'uscita Q si collega alla porta AND che genera il segnale START RIBBON MOTION PULSE e alla logica d'innesco del monostabile 555; in nessuno dei due casi la transizione alto-basso di Q ha effetto. Il segnale

impulsivo START RIBBON MOTION è già basso e il monostabile 555 è innescato da una transizione di Q alto-basso. La transizione basso-alto porta il segnale di "trigger" ad un livello alto che non richiede alcuna simulazione.

L'uscita ($\overline{Q}$) di FFC è messa in AND con l'uscita $\overline{Q}$ del monostabile PW RELEASE ENABLE per generare l'ingresso (K) di FFD. L'ingresso (J) di FFD proviene direttamente da (Q) di FFC, perciò **non appena il monostabile PW RELEASE ENABLE va nuovamente alto, FFD riceverà un basso sull'ingresso J ed un alto sull'ingresso K:**

Un ingresso J basso ed un ingresso K alto sul flip-flop FFD lo porta nello stato "off" e ciò innesca il monostabile PW READY ENABLE.

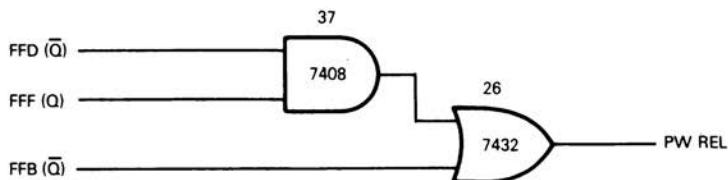
SIMULAZIONE DEL MONOSTABILE PW READY ENABLE

La logica associata a questo monostabile è quasi identica a quella del monostabile PW RELEASE ENABLE. La commutazione nello stato "off" di FFD provoca una transizione basso-alto sulla sua uscita $\overline{Q}$, che innesca il monostabile PW READY ENABLE.

Dobbiamo ora simulare un ritardo di tempo di due millisecondi; la prossima sequenza di istruzioni è quasi identica alla simulazione del monostabile PW RELEASE ENABLE e può essere illustrata come segue:

```
; Esecuzione di un ritardo di tempo di 3 millisecondi
        LD   DE,230      ; Carica la costante di tempo in D, E
PWRL:   DEC  DE          ; Decremento della coppia di registri
        LD   A,D         ; Test per lo zero
        OR   E
        JR   NZ,PWRL     ; Ridecrementa se non è zero
```

```
; Commutazione nello stato "off" del flip-flop FFD
          IN    A,(4)        ; Posiziona a 0 il bit 3 della Porta di I/O A1
          RES   3,A
          OUT   (4),A
; Esecuzione di un ritardo di tempo di due millisecondi
          LD    A,250        ; Carica la costante di tempo iniziale nell'Accu-
                             ; mulatore
PWRD:     DEC   A            ; Decremento dell'Accumulatore
          JR    NZ,PWRD      ; Ridecrementa se non è zero
```

Quando FFD commuta nello stato "off", l'uscita PW REL va di nuovo alta. Ecco la costruzione logica di PW REL:

L'uscita ($\overline{Q}$) è ancora bassa in quest'istante. Ma l'uscita ($\overline{Q}$) di FFD e l'uscita (Q) di FFF sono entrambe alte, così la porta AND 37 ha un impulso alto in uscita che passa attraverso la porta OR 26 per posizionare alto PW REL.

Queste istruzioni posizionano alto PW REL:

```
; Esecuzione di un ritardo di tempo di due millisecondi
          LD    A,250        ; Carica la costante di tempo iniziale nell'Accu-
                             ; mulatore
PWRD:     DEC   A            ; Decremento dell'Accumulatore
          JR    NZ,PWRD      ; Ridecrementa se non è zero
; Posizionamento di PW REL ad alto
          IN    A,(2)        ; La Porta di I/O B0 entra nell'Accumulatore
          SET   0,A          ; Posiziona ad 1 il bit 0
          OUT   (2),A        ; Ritorno del risultato
```

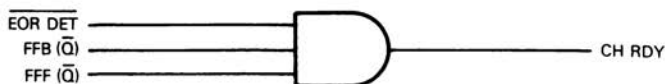
Ora l'intero ciclo di stampa finisce in fretta. Le uscite Q e $\overline{Q}$ del flip-flop FFD diventano gli ingressi J e K di FFE. Q è dapprima messa in AND con FFI, che, a sua volta, è costantemente alto; perciò quando FFD va "off", FFE riceve un basso sull'ingresso J.

L'ingresso (K) di FFE non va alto finchè non finisce il colpo del monostabile PW READY ENABLE, poichè l'uscita $\overline{Q}$ di PW READY ENABLE è messa con $\overline{Q}$ di FFD per generare l'ingresso (K) di FFE.

La commutazione nello stato "off" di FFE è il nostro prossimo evento cronologico.

La commutazione nello stato "off" di FFE, a sua volta, provoca la commutazione nello stato "off" di FFB e di FFF, FFB è messo nello stato "off" dalla transizione basso-alto dell'uscita ($\overline{Q}$) di FFE, che diventa l'ingresso di clock di FFB. FFF commuta nello stato "off" poichè i suoi ingressi J e K sono collegati direttamente alle uscite Q e $\overline{Q}$ di FFE.

Una volta che FFB e FFF sono commutati nello stato "off", si verificano tutte le condizioni per cui CH RDY può andare di nuovo alto, stabilendo che $\overline{\text{EOR DET}}$ non

segnali la fine del nastro:

```
; Esecuzione di un ritardo di tempo di due millisecondi
         LD    A,250       ; Carica la costante di tempo iniziale nell'Accu-
                           ; mulatore
PWRD:    DEC   A           ; Decremento dell'Accumulatore
         JR    NZ,PWRD     ; Ridecrementa se non è zero
; Posizionamento di PW REL ad alto
         IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
         SET   0,A         ; Posiziona ad 1 il bit 0
         OUT   (2),A       ; Ritorno del risultato
; Commutazione nello stato "off" dei flip-flop FFB, FFE e FFF
         IN    A,(4)       ; La Porta A1 entra nell'Accumulatore
         AND   0AFH        ; Posiziona a 0 i bit 4 e 6
         OR    22          ; Posiziona ad 1 i bit 5 e 1
         OUT   (4),A       ; Uscita del risultato
; Posizionamento ad alto di CH RDY
         IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
         SET   1,A         ; Posizionamento ad 1 del bit 1
         OUT   (2),A       ; Uscita del risultato
; Salto al test di fine valida del ciclo di stampa
         JP    VALND
```

SOMMARIO DELLA SIMULAZIONE

Il completo programma di simulazione sviluppato in questo Capitolo è dato in Figura 3-3. I numeri cerchiati corrispondono ai numeri di Figura 3-2.

Possiamo concludere che una simulazione uno a uno, assolutamente esatta della logica digitale usando istruzioni in linguaggio assembly in un sistema a microcalcolatore non è fattibile; ma allora non è particolarmente desiderabile.

Se non siete progettisti logici digitali, sarete stati molto confusi dalle varie combinazioni di segnali richieste nella logica di Figura 3-1. Gran parte di ciò che verrà trattato non avrà a che fare con i fondamentali requisiti della stampante Qume; piuttosto essa rispecchia una implementazione logica interna di un progettista di logica, diretta ad assicurare sequenze di segnali esterni appropriate in tutte le circostanze concepibili.

Se siete progettisti logici, ci sono possibilità che implementiate i requisiti specifici dell'interfaccia con la stampante Qume in maniera totalmente differente; potreste anche brontolare per questa implementazione.

Il punto importante da tenere in mente è che la logica digitale contiene innumerevoli sottigliezze che sono specifiche dei dispositivi logici discreti. Queste sottigliezze non sono collegate ai requisiti della implementazione totale.

Il linguaggio assembly ha il proprio insieme di sottigliezze, che pure esse non hanno niente a che fare con la implementazione finale; piuttosto esse sono dirette a fare un uso più efficiente delle singole istruzioni o delle sequenze di istruzioni.

Non dovrebbe perciò sorprendere che una esatta duplicazione di logica digitale, usando un linguaggio assembly, non sia né fattibile né desiderabile. Così ci allontaneremo dalla logica digitale e cominceremo a trattare un problema dal punto di vista della programmazione.

LINGUAGGIO ASSEMBLY RISPETTO A LOGICA DIGITALE

La differenza principale tra logica digitale e linguaggio assembly è che il linguaggio assembly tratta gli eventi cronologicamente, mentre la logica digitale isola la logica in nodi funzionali. In tale modo un dispositivo logico può essere responsabile di un numero di eventi che accadono in istanti diversi durante un ciclo logico; quando è trasferito in linguaggio assembly, ogni evento diventa una sequenza di istruzioni isolata.

In Figura 3-1, per esempio, il ciclo di stampa cominciava con la commutazione nello stato "on" di flip-flop in cascata e finiva con la commutazione nello stato "off" degli stessi flip-flop. In molti casi un flip-flop commutando nello stato "on" avvia un evento, mentre, commutando lo stesso flip-flop nello stato "off", avvia un evento completamente differente. In un programma in linguaggio assembly i due eventi non avranno niente in comune. Ogni evento sarà rappresentato da una sequenza di istruzioni completamente indipendenti che capita in parti completamente differenti del programma.

La maggiore differenza tra logica digitale e linguaggio assembly è il concetto di temporizzazione. In un sistema a logica digitale sincrono, come illustrato in Figura 3-1, la temporizzazione è limitata ai segnali di clock e alla necessità di interazioni pulite di segnali. In un programma in linguaggio assembly, la temporizzazione dipende rigidamente dalla sequenza di esecuzione delle istruzioni. Inoltre, mentre circuiti in logica digitale possono commutare e funzionare in parallelo, in un programma in linguaggio assembly ogni cosa deve accadere serialmente.

Il concetto chiave da afferrare in questo capitolo è che non c'è niente di istintivamente corretto circa la logica digitale come mezzo per implementare qualcosa. Il fatto che non siamo stati capaci di duplicare esattamente la logica digitale usando istruzioni in linguaggio assembly non significa che il linguaggio assembly sia in qualche modo inferiore, significa semplicemente che il linguaggio assembly farà il lavoro in modo differente.

Dopo aver speso il nostro tempo nel Capitolo 3 per disegnare parallelismi diretti tra linguaggio assembly e logica digitale, abbandoneremo ora ogni tentativo di favorire la logica digitale. Nel Capitolo 4 la logica illustrata in Figura 3-1 sarà simulata di nuovo — ma dal punto di vista del programmatore.

```
 Assegnazione di locazioni per l'inizio della tabella di conteggio dei ritardi DELY
; EQU NNNNH
; Test sulla validità della fine di un ciclo di stampa
VALND:
          IN    A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
          RLA                  ; Sposta il bit 7 nel Carry
          JR    NC,VALND       ; Se il Carry è zero, rimanere nel ciclo di stampa
; Esecuzione del programma tra due cicli di stampa
; Posizionamento iniziale a 1 dei bit 5, 1 e 0 ed a 0 dei bit 6, 4, 3 e 2 della Porta
; di I/O A1
          IN    A,(4)          ; La Porta di I/O A1 entra nell'Accumulatore
```

Figura 3-3. Il programma di simulazione completo (segue)

```
           OR     23H            ; Posizionamento ad 1 dei bit 5, 1 e 0
           AND    0A3H           ; Posizionamento a 0 dei bit 6, 4, 3 e 2
           OUT    (4),A          ; Ritorno del risultato
 Test su RETURN STROBE basso
STBHI:     IN     A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
           BIT    4,A            ; Test sul bit RETURN STROBE
           JR     Z,FFB          ; Se è 0, saltare alla simulazione di FFB
; Simulazione di FFAW e della logica associata
; Carica il contenuto della Porta di I/O B0 nell'Accumulatore ed isola i bit 1, 5 e 6
; per CH RDY, PW STROBE e RESET, rispettivamente
①          IN     A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
           AND    62H            , Isola i bit 6, 5 e 1. Se RESET = 0
           CP     22H            ; CH RDY = 1 e PW STROBE = 1, comincia il primo
                                 ; ciclo di stampa
           JR     NZ,STBHI       ; Altrimenti si torna a STBHI
②          IN     A,(4)          ; Inizio di un nuovo ciclo di stampa posizionando a 0
           RES    0,A            ; Il bit 0 della Porta di I/O A1
           OUT    (4),A
; La sequenza di istruzioni del nuovo ciclo di stampa comincia qui
; Simulazione della commutazione nello stato "on" del flip-flop FFB

FFB:       IN     A,(4)          ; Carica la Porta di I/O A1 nell'Accumulatore
③          RES    1,A            ; Posiziona a 0 il bit 1
           OUT    (4),A          ; Memorizza il risultato
; Simulazione della porta AND 7411 che posiziona basso CH RDY. Inoltre la porta
; OR 7432 mette PW REL basso
⑤          IN     A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
           AND    0FCH           ; Posiziona a 0 i bit 0 ed 1
           OUT    (2),A          ; Memorizza il risultato
; CH RDY basso mette FFA nello stato "off". Posiziona ad 1 il bit 0 della Porta di
; I/O A1 inoltre simula la commutazione nello stato "on" di FFC e di FFD. Posi-
; ziona i bit 2 e 3 della Porta di I/O A1
⑥          IN     A,(4)          ; Carica la Porta di I/O A1 nell'Accumulatore
⑧          OR     0DH            ; Posiziona i bit 3, 2 e 0 ad 1
           OUT    (4),A          ; Memorizza il risultato
; Impulso START RIBBON MOTION alto
⑦          IN     A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
           SET    3,A            ; Posizionamento del bit ad 1
           OUT    (2),A          ; Uscita verso la Porta di I/O B0
           RES    3,A            ; Posizionamento del bit 3 a 0
           OUT    (2),A          ; Uscita verso la Porta di I/O B0
; Prova sull'ingresso della decodifica della velocità per creare un ritardo al moto della
; ruota di stampa
VLDC:      IN     A,(0)          ; La Porta di I/O A0 entra nell'Accumulatore
           RLA                   ; Sposta il bit 7 nel Carry
           JR     NC,VLDC        ; Rimane in loop se il Carry è zero
; Alla fine del ritardo si simula la commutazione nello stato "on" di FFE
⑨          IN     A,(4)          ; La Porta di I/O A1 entra nell'Accumulatore
           RES    5,A            ; Posiziona a 0 il bit 5
           SET    4,A            ; Posiziona ad 1 il bit 4
           OUT    (4),A          ; Uscita del risultato
 Simulazione del ritardo di tempo di 2 ms per PW SETTLING
```

Figura 3-3. Il programma di simulazione completo (segue)

```
        LD    A,0FAH      ; Carica la costante iniziale del ritardo di tempo
PWS:    DEC   A           ; Decrementa l'Accumulatore
        JR    NZ,PWS      ; Ridecrementa se il risultato non è zero
; Simulazione della commutazione nello stato "on" del flip-flop FFF
FFF:    IN    A,(0)       ; La Porta di I/O A0 entra nell'Accumulatore
        CPL               ; Complemento per verificare se qualche bit è ad 1
        AND   1FH         ; Isola i bit da 0 a 4
        JR    NZ,FFF      ; Se qualche bit è 1, rimane in loop
        IN    A,(4)       ; Posizionamento a 1 del bit 6 della Porta di I/O A1
⑩       SET   6,A
        OUT   (4),A
; Test su HAMMER ENABLE FF
        IN    A,(0)       ; La Porta di I/O A0 entra nell'Accumulatore
        BIT   6,A         ; Test sul bit 6
        JR    Z,HP0       ; Se zero, si supera il posizionamento a basso di
                          ; HAMMER PULSE
; HAMMER ENABLE FF è alto, così HAMMER PULSE deve essere messo basso perciò
; si posiziona a 0 il bit 2 della Porta di I/O B0
⑪       IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        RES   2,A         ; Posiziona a 0 il bit 2
        OUT   (2),A       ; Uscita del risultato
; Calcolo del ritardo di tempo
HP0:    LD    HL,DELY     ; Caricamento dell'indirizzo di base del ritardo nella
                          ; coppia HL
        IN    A,(6)       ; La Porta B1 (selettore) entra nell'Accumulatore
HP1:    RRA               ; Rotazione a destra dell'Accumulatore attraverso
                          ; il Carry
        INC   HL          : Incrementa di 2 il contenuto di HL
        JR    NC,HP1      ; Se non c'è Carry, ruotare ed incrementare di nuovo
        LD    E,(HL)      ; Carica in DE la costante a 16 bit del ritardo di tempo
        INC   HL
        LD    D,(HL)
TDLY:   DEC   DE          ; Esecuzione del loop del ritardo di tempo
        LD    A,D
        OR    E
        JR    NZ,TDLY
; Uscita ancora alta su HAMMER PULSE
        IN    A,(2)       ; La Porta di I/O B0 entra nell'Accumulatore
        SET   2,A         ; Posiziona ad 1 il bit 2
        OUT   (2),A       ; Uscita del risultato
; Commutazione nello stato "off" del flip-flop FFC
⑫       IN    A,(4)       ; Posizionamento a 0 del bit 2 della porta di I/O A1
        RES   2,A
        OUT   (4),A
; Esecuzione di un ritardo di tempo di 3 millisecondi
        LD    DE,230      ; Carica la costante di tempo in D,E
PWRL:   DEC   DE          ; Decrementa la coppia di registri
        LD    A,D         ; Test per zero
        OR    E
        JR    NZ,PWRL     ; Ridecrementa se non è zero
 Commutazione nello stato "off" del flip-flop FFD
⑬       IN    A,(4)       ; Posizionamento a 0 del bit 3 della Porta di I/O A1
```

Figura 3-3. Il programma di simulazione completo (segue)

```
        RES   3,A
        OUT   (4),A
; Esecuzione del ritardo di tempo di 2 millisecondi
        LD    A,250          ; Carica la costante di tempo iniziale nell'Accu-
                             ; mulatore
PWRD:
        DEC   A              ; Decrementa l'Accumulatore
        JR    NZ,PWRD        ; Ridecrementa se non è zero
; Posizionamento ad alto di PW REL
⑭       IN    A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
        SET   0,A            ; Posizionamento ad 1 del bit 0
        OUT   (2),A          ; Ritorno del risultato
; Commutazione nello stato "off" dei flip-flop FFB, FFE e FFF
⑮       IN    A,(4)          ; La Porta di I/O A1 entra nell'Accumulatore
        AND   0AFH           ; Posizionamento a 0 dei bit 4 e 6
⑯       OR    22             ; Posizionamento ad,1 dei bit 5 e 1
        OUT   (4),A          ; Uscita del risultato
 Posizionamento ad alto di CH RDY
⑰       IN    A,(2)          ; La Porta di I/O B0 entra nell'Accumulatore
        SET   1,A            ; Posizionamento ad 1 del bit 1
        OUT   (2),A          ; Uscita del risultato
, Salto al test per la fine valida del ciclo di stampa
        JP    VALND
 ;Tabella dei contatori dei ritardi
        ORG   DELY+2
        DEFW PPQQH           : Ritardo di tempo H1
        DEFW RRSSH           ; Ritardo di tempo H2
        DEFW TTUUH           ; Ritardo di tempo H3
        DEFW VVWWH           ; Ritardo di tempo H4
        DEFW XXYYH           ; Ritardo di tempo H5
        DEFW ZZOOH           ; Ritardo di tempo H6
```

Figura 3-3. Il programma di simulazione completo (conclusione)

Capitolo 4

UN PROGRAMMA SEMPLICE

I problemi associati alla simulazione di logica digitale, come abbiamo fatto nel Capitolo 3, possono essere attribuiti ad un solo fatto: abbiamo cercato di dividere la logica in un numero di funzioni di trasferimento isolate, ognuna delle quali corrispondeva ad un dispositivo logico digitale. Abbandoneremo ora la logica digitale e combinatoria supponendo che essa non esista e guarderemo in altro modo le Figure 3-1 e 3-2.

TEMPORIZZAZIONE IN LINGUAGGIO ASSEMBLY RISPETTO A TEMPORIZZAZIONE IN LOGICA DIGITALE

FUNZIONE DI TRASFERIMENTO

Tornando alla Figura 3-1, ignoriamo semplicemente tutto ciò che esiste tra i bordi di sinistra e di destra della figura. Ciò che rimane è un insieme di segnali d'ingresso e un insieme di segnali di uscita. I segnali di uscita sono collegati ai segnali d'ingresso da un insieme di funzioni di trasferimento che non hanno nulla a che fare con dispositivi logici digitali.

Le funzioni di trasferimento di Figura 3-1 sono liberamente rappresentati dal diagramma di temporizzazione di Figura 3-2. Che cosa significa "rappresentati liberamente"? Significa che la temporizzazione che ha relazione con i requisiti del sistema è mescolata indiscriminatamente con la temporizzazione che riflette semplicemente le necessità della logica digitale. Possiamo tralasciare le considerazioni di temporizzazione che rispecchiano semplicemente le necessità della logica digitale. Per essere specifici, il martelletto di stampa deve essere alimentato inviando un solo impulso su sei possibili al solenoide; i vari ritardi dei movimenti e dei posizionamenti devono essere mantenuti. Ma possiamo abbandonare i ritardi di tempo che separano un cambiamento dello stato di un segnale da un altro semplicemente per tenere pulita la logica digitale.

Dal punto di vista del programmatore, tuttavia, il diagramma di temporizzazione illustrato in Figura 4-1 è una sostituzione perfettamente valida per un progettista logico del diagramma di temporizzazione illustrato in Figura 3-2.

SEGNALI DI INGRESSO E DI USCITA

Guardando la Figura 4-1 vedrete che abbiamo abbandonato una certa quantità di ritardi di temporizzazione; inoltre abbiamo tralasciato la maggior parte dei nostri segnali. Ma esiste un semplice criterio per determinare se un segnale in un sistema a microcalcolatore è realmente necessario. Il criterio è questo: se il segnale è associato unicamente ad eventi di tempo reali della logica esterna al sistema del microcalcolatore, allora il segnale deve rimanere. Se la sorgente e la destinazione del segnale sono nella "scatola nera" del sistema del microcalcolatore, allora il segnale può essere abbandonato. Basandoci su questo criterio, guardiamo di nuovo i nostri segnali di ingresso e di uscita.

Cominciamo dapprima con i segnali di ingresso.

SEGNALI DI INGRESSO

RETURN STROBE e PW STROBE sono segnali senza significato. Come logica digitale, questi due segnali danno inizio alla sequenza del ciclo di stampa. In un programma in linguaggio assembly, tutto ciò che si richiede per la inizializzazione è di saltare alla prima istruzione della sequenza. Il fatto che RETURN STROBE rappresenti un ciclo di stampa durante il quale non si alimenta il martelletto di stampa non è importante, perchè per sopprimere realmente HAMMER PULSE si usa HAMMER ENABLE. **Combineremo i vari segnali che impediscono l'alimentazione del martelletto in un solo stato di ingresso del martelletto.** Ci sono cinque segnali interessati: PFL REL, RIB LIFT RDY, RIBBON ADVANCE, PFR REL e CA REL. Ognuno di questi segnali ha origine da logica esterna differente alla Figura 3-1; nella implementazione della logica digitale, questi segnali sono messi in AND per creare un segnale principale HAMMER INTERLOCK. Nella nostra implementazione in linguaggio assembly metteremo in OR cablato tutti questi segnali esterni su un singolo pin che diventa uno stato HAMMER INTERLOCK.

RESET rimarrà come un segnale principale di Reset collegato al pin RESET della CPU. RESET può quindi essere ignorato dal programma in linguaggio assembly; tuttavia ricordate che ogni volta che RESET è attivato, l'esecuzione del programma riprenderà dall'istruzione memorizzata nella locazione 0 della memoria.

EOR DET sarà mantenuto. Questo è il segnale che rileva la fine del nastro ed impedisce che un ciclo di stampa non abbia mai fine, impedendo la stampa di ulteriori caratteri dopo l'esaurimento del nastro.

HAMMER ENABLE FF deve essere conservato; esso sopprime l'impulso di alimentazione del martelletto di stampa durante i cicli di stampa di riposizionamento della ruota di stampa.

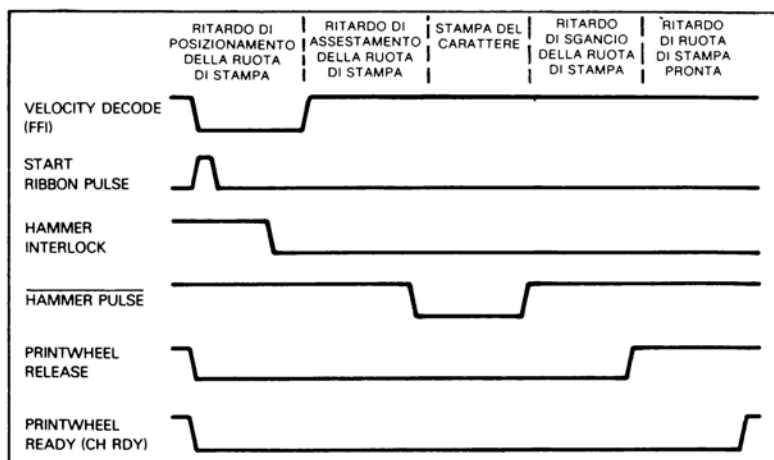



Figura 4-1. Temporizzatore della Figura 3-1, dal punto di vista del programmatore

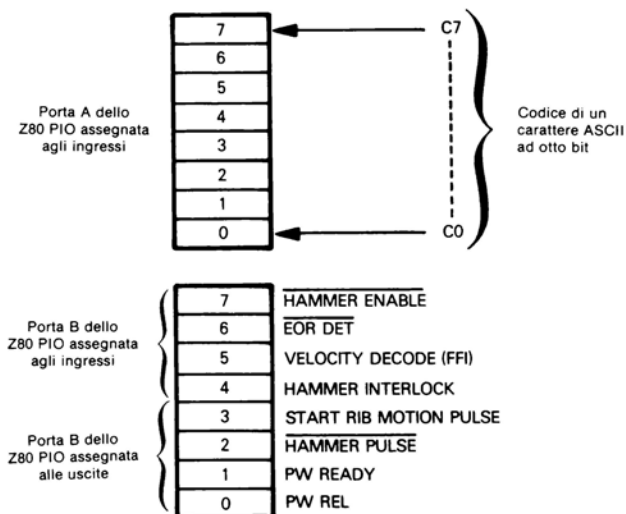
La funzione realizzata dai sei segnali della lunghezza dell'impulso del martelletto, da H1 a H6, deve essere conservata, ma i segnali scompariranno. Invece di usare sei pin di una porta di I/O per identificare la durata dell'impulso del martelletto, creeremo ritardi di tempo direttamente dai codici in carattere ASCII.

Prestiamo ora attenzione ai segnali di uscita.

Per cominciare, possiamo eliminare tutte le uscite dei flip-flop. Il comportamento di ogni intervallo di tempo in un ciclo di stampa è già identificato da un esistente stato di cambiamento del segnale. Se si deve innescare più di un evento logico esterno con una transizione da un intervallo di tempo al successivo, non c'è niente che impedisca al segnale appropriato di essere memorizzato esternamente, quindi dall'essere usato per innescare numerosi eventi logici esterni. In un programma di microcalcolatore, non c'è motivo di duplicare dei segnali che dovrebbero essere fatti uscire semplicemente per identificare la transizione da un intervallo di tempo di un ciclo di stampa al successivo.

Rimanenti segnali di uscita sono mantenuti. E' possibile che alcuni di questi segnali scompaiano se la logica esterna addizionale sarà sostituita da più programmi in linguaggio assembly nel sistema del microcalcolatore; ma, dati i limiti del problema, come stabilito, i rimanenti segnali sono necessari per definire gli intervalli di tempo dei cicli di stampa.

ASSEGNAZIONE DEI PIEDINI (PIN)

Con un insieme di segnali nuovi e semplificati, possiamo eliminare uno Z80 PIO; per il rimanente Z80 PIO, le porte di I/O e i pin sono assegnati come segue:

CONFIGURAZIONE DEI DISPOSITIVI DEL MICROCALCOLATORE

Siamo ora nella posizione di scegliere i dispositivi necessari per l'implementazione del programma. La scelta in realtà è molto chiara: in aggiunta alla CPU, abbiamo

bisogno di un dispositivo Z80 PIO (Parallel Input/Output), di un po' di memoria a sola lettura (ROM) per la memorizzazione del programma, e di un po' di memoria di lettura e scrittura (RAM) per memorizzare i dati generali. La Figura 4-2 illustra il sistema del microcalcolatore che risulta dopo aver combinato questi dispositivi. Ora se non capite immediatamente la Figura 4-2 non disperate; ci sono solo alcuni aspetti in questa figura che hanno conseguenze sulla nostra discussione immediata.

CONCETTI GENERALI DI PROGETTAZIONE

Il più importante concetto che deriva dalla Figura 4-2 è questo: quando si progetta della logica scrivendo dei programmi in linguaggio assembly in un sistema a microcalcolatore, il programma che scriverete diventerà altamente dipendente dalla configurazione dei dispositivi. Il modo col quale i dispositivi di Figura 4-2 sono stati combinati non è l'unico; sono parimenti fattibili configurazioni alternative. Tuttavia,

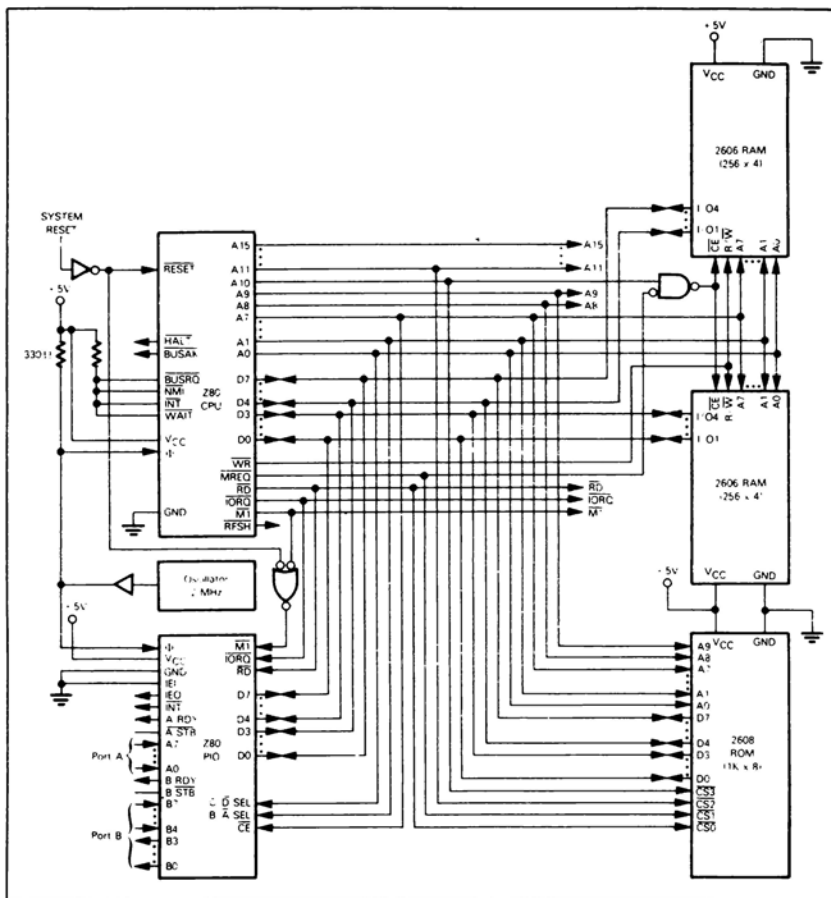



Figura 4-2. Configurazione del microcalcolatore con lo Z80

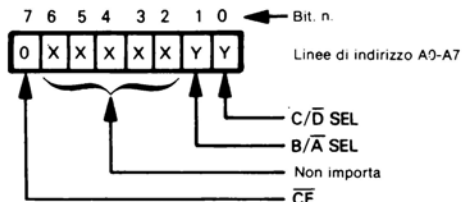
i programmi in linguaggio assembly creati, possono differire notevolmente da una configurazione di microcalcolatore ad un'altra, e questo è un fattore da non perdere di vista quando si scrivono programmi su microcalcolatori. Inoltre non abbiate paura di modificare la configurazione di hardware scelta. **La configurazione dei dispositivi del microcalcolatore e la programmazione in linguaggio assembly interagiscono fortemente e non dovrebbero essere separati.** Questi due passi dovrebbero essere in un loop iterativo. Durante i primi stadi della scrittura di un programma di microcalcolatore, dovreste supporre che nel corso della scrittura di un programma in linguaggio assembly potrete scoprire aspetti dell'hardware che possono essere migliorati; ciò significa che anche i programmi dovranno essere riscritti.

LINGUAGGI AD ALTO LIVELLO

Questo è un buon punto al quale fa riferimento uno dei motivi che rendono i linguaggi ad alto livello non desiderabili quando si programma un microcalcolatore per sostituire logica digitale. I linguaggi ad alto livello sono orientati al problema. Per esempio, è difficile guardare uno "statement" di un programma in PL/M e visualizzare il modo esatto con cui i dati saranno mossi in un sistema a microcalcolatore in risposta alla esecuzione dello "statement". E' ancora più duro riferire programmi in PL/M a precise configurazioni di dispositivi. Il linguaggio assembly, d'altra parte, ha una relazione a uno a uno col vostro hardware.

L'INTERFACCIA PARALLELA DI INGRESSO/USCITA (PIO) DELLO Z80

Torniamo al modo specifico in cui si sono incorporati i dispositivi nella Figura 4-2.

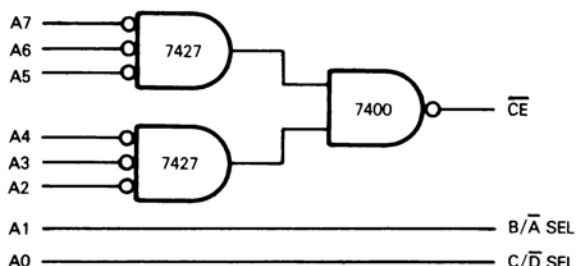
Lo Z80 PIO risponderà ad indirizzi di I/O come segue:

SELEZIONE DEI CHIP NEI SISTEMI SEMPLICI

Supporremo che tutti i bit di indirizzo "don't care" siano 0; come risultato **useremo i quattro indirizzi delle porte di I/O da 0 a 3 per indirizzare il singolo Z80 PIO di Figura 4-2. I quattro indirizzi accederanno alle locazioni del Z80 PIO come segue:**

0 : Dati della Porta A
1 : Controllo della Porta A
2 : Dati della Porta B
3 : Controllo della Porta B

Se la configurazione del microcalcolatore contiene un grande numero di porte di I/O, la logica di selezione del chip diventa un po' più complessa. Se uno Z80 PIO deve rispondere esattamente a quattro indirizzi di una porta di I/O, escludendo tutti gli altri, allora la selezione del chip deve essere creata combinando in un unico modo le otto linee di indirizzo di peso minore.

SELEZIONE DEI CHIP NEI SISTEMI PIU' GRANDI

Supponiamo che lo Z80 PIO di Figura 4-2 debba rispondere solo agli indirizzi da 0 a 3 della porta di I/O. Ora tutte le linee di segnale "don't care" devono essere ingressi per la logica che è vera solo quando queste linee di segnale sono tutte basse. **Ecco un modo per creare la logica di selezione del chip:**

L'ingresso $\overline{CE}$ può essere creato usando due delle tre porte di una porta NOR 7427 tripla e a 3 Ingressi Positivi ed una delle quattro porte della porta NAND 7400 Quadrupla e a 2 Ingressi.

Data la logica di selezione precedente, lo Z80 PIO si considererà selezionato se, e solo se, si fa uscire sul Bus degli Indirizzi uno dei quattro indirizzi specificati della porta di I/O.

La direzione dei dati e la utilizzazione della porta illustrata per lo Z80 PIO in Figura 4-2 non hanno caratterizzazione in hardware. La utilizzazione della porta può essere modificata in ogni istante scrivendo le appropriate parole di controllo nei registri di Controllo dello Z80 PIO.

LOGICA DI RESET

La logica di Reset ha bisogno di un commento. Invece di fare dei test su una condizione di Reset tra due cicli di stampa, come si è fatto nel Capitolo 3, useremo un segnale di Reset hardware, ma in un contesto con microcalcolatore.

LOGICA DI RESET DELLO Z80 PIO

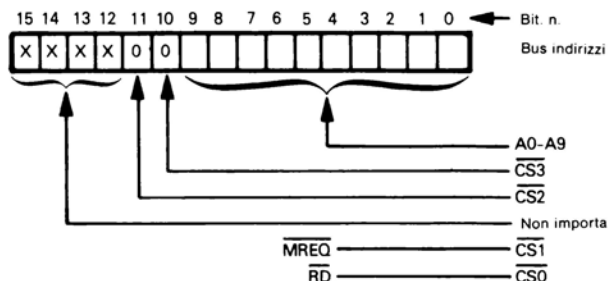
Il segnale $\overline{RESET}$ è collegato all'ingresso $\overline{M1}$ dello Z80 PIO. Quando lo Z80 PIO riceve $\overline{M1}$ basso mentre $\overline{RD}$ e $\overline{IORQ}$ sono entrambi bassi, esso è azzerato col Reset; entrambi le porte sono nel Modo 1 - ingresso con "handshaking". In un punto dopo il Reset hardware, il programma di CPU deve posizionare lo Z80 PIO per il nostro particolare scopo eseguendo istruzioni che scrivono le appropriate parole di controllo.

L'attivazione dell'ingresso di $\overline{RESET}$ dello Z80 CPU azzererà il Contatore di Programma (Program Counter); ciò significa che l'esecuzione del programma comincerà con l'istruzione memorizzata nel byte di memoria il cui indirizzo è 0. Dobbiamo perciò avere un programma di "dopo reset" e di inizializzazione del sistema che inizi da questa locazione di memoria.

La logica di selezione della memoria illustrata in Figura 4-2 soddisferà i requisiti della logica di Reset.

MEMORIA ROM E RAM

INDIRIZZI DELLE ROM

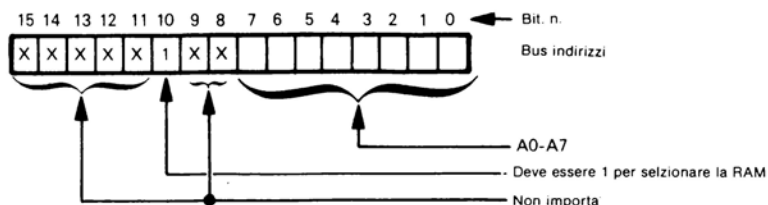
Una 2608 della Signetics fornisce al nostro sistema di microcalcolatore 1024 byte di memoria a sola lettura (ROM). Due delle quattro linee di selezione, più dieci linee di indirizzo, creano gli indirizzi della ROM come segue:

Le altre due linee di selezione sono collegate ai segnali di controllo $\overline{\text{MREQ}}$ e $\overline{\text{RD}}$. Se i bit di indirizzo "don't care" si suppongono uguali a 0, allora la memoria ROM sarà selezionata dagli indirizzi da 0 a $03FF_{16}$. Questa fornisce il byte di memoria che dobbiamo avere all'indirizzo 0 quando lo Z80 CPU comincia ad eseguire le istruzioni dopo un Reset.

Notate che in nessun caso lo spazio di indirizzo della ROM entrerà in conflitto con gli indirizzi dello Z80 PIO: $\overline{\text{MREQ}}$ deve essere attivato per scegliere la ROM 2608, mentre $\overline{\text{IORQ}}$ deve essere attivato per scegliere lo Z80 PIO. Lo Z80 CPU non attiva mai $\overline{\text{MREQ}}$ e $\overline{\text{IORQ}}$ contemporaneamente.

SELEZIONE DELLA ROM IN SISTEMI SEMPLICI

Ancora una volta usiamo una primitiva selezione del chip di ROM a causa della semplicità del sistema del microcalcolatore. Abbiamo definito il campo degli indirizzi per i 1024 byte della memoria ROM variabile da 0 a $03FF_{16}$; ma, in realtà, una grande varietà di altri indirizzi accedono pure alla memoria — le linee di indirizzo da A12 ad A15 possono avere un valore qualsiasi. Stabilendo che A10 ed A11 siano entrambi 0, si avrà accesso alla memoria ROM. Non c'è nulla che vi impedisca di scegliere la memoria in questo modo primitivo, stabilendo che il vostro sia un piccolo sistema a microcalcolatore. Non c'è alcuna ragione perchè dobbiate incorrere in spese addizionali creando codici di selezione della memoria complessi usando tutte le sei linee di ordine maggiore del bus degli indirizzi. **Anche dal punto di vista del programmatore, non dovrete riscrivere programmi in caso di espansione del vostro sistema e di inclusione di maggiore memoria in un secondo tempo. Stabilendo ora che non userete nessuno degli indirizzi alternativi che pure selezionerebbero la ROM, allora in un tempo futuro potrete prendere uno di questi insiemi di alternative di indirizzi, usarlo per selezionare un'altra ROM e non influenzare in alcun modo i programmi già scritti.**

Specificando la ROM di memorizzazione del programma, supponiamo che il prodotto sarà sviluppato in un volume sufficiente per giustificare la spesa per creare una maschera per la ROM. Se il vostro volume non giustifica la spesa per creare una ROM, allora voi potete usare una PROM (Programmable Read Only Memory), memoria a sola lettura programmabile.

Le due RAM 2606 Signetics forniscono ciascuna 1024 bit di memoria a lettura scrittura, organizzata in 256 unità di 4 bit. Ogni RAM perciò fornisce la metà del byte della memoria di lettura e scrittura. I 256 byte della RAM avranno gli indirizzi da 0400_{16} a $04FF_{16}$. Ciò può essere illustrato come segue:

Anche se abbiamo specificatò gli indirizzi da 0400_{16} a $04FF_{16}$ come indirizzi della RAM, ancora una volta un grande numero di altri indirizzi selezionerebbero la RAM. Notate, tuttavia, che in nessun caso un indirizzo della RAM coinciderà con un indirizzo della ROM; la linea A10 del Bus degli Indirizzi deve essere sempre 0 per selezionare la ROM, mentre deve essere sempre 1 per selezionare la RAM. Non ci saranno mai, perciò, contese sugli indirizzi.

In sommario, gli indirizzi del sistema di microcalcolatore illustrato in Figura 4-2 saranno interpretati come segue:

	INDIRIZZI	
PORTE DI I/O	00_{16} – 03_{16}	Z80 PIO
MEMORIA	0000_{16} – $03FF_{16}$	Memoria a sola lettura
MEMORIA	0400_{16} – $04FF_{16}$	Memoria a lettura e scrittura

INIZIALIZZAZIONE DEL SISTEMA

Volgiamo ora la nostra attenzione alle operazioni del sistema.

Quando si inizializza il sistema, si devono ristabilire immediatamente le condizioni "tra due cicli di stampa". I passi necessari sono questi:

1) Se è stato alimentato il martelletto di stampa, interrompere l'impulso di alimentazione e concedere tempo al martelletto di stampa di ritirarsi.

2) Spostare la ruota di stampa indietro nella sua posizione di visibilità.

3) Assicurarsi che i segnali di uscita abbiano il loro stato "tra due cicli di stampa".

SEQUENZA DI IMPLEMENTAZIONE DEL PROGRAMMA

Giungiamo ora ad un altro fondamentale concetto di programmazione: **esiste una sequenza "la più efficiente" secondo la quale si potrebbero scrivere programmi sorgenti in linguaggio assembly.** Potremmo andare avanti e scrivere un programma di inizializzazione che implementi un Reset, ma ciò richiederebbe una grande quantità di supposizioni. Come possiamo sapere che si è appena alimentato il martelletto di stampa? Come spostiamo la ruota

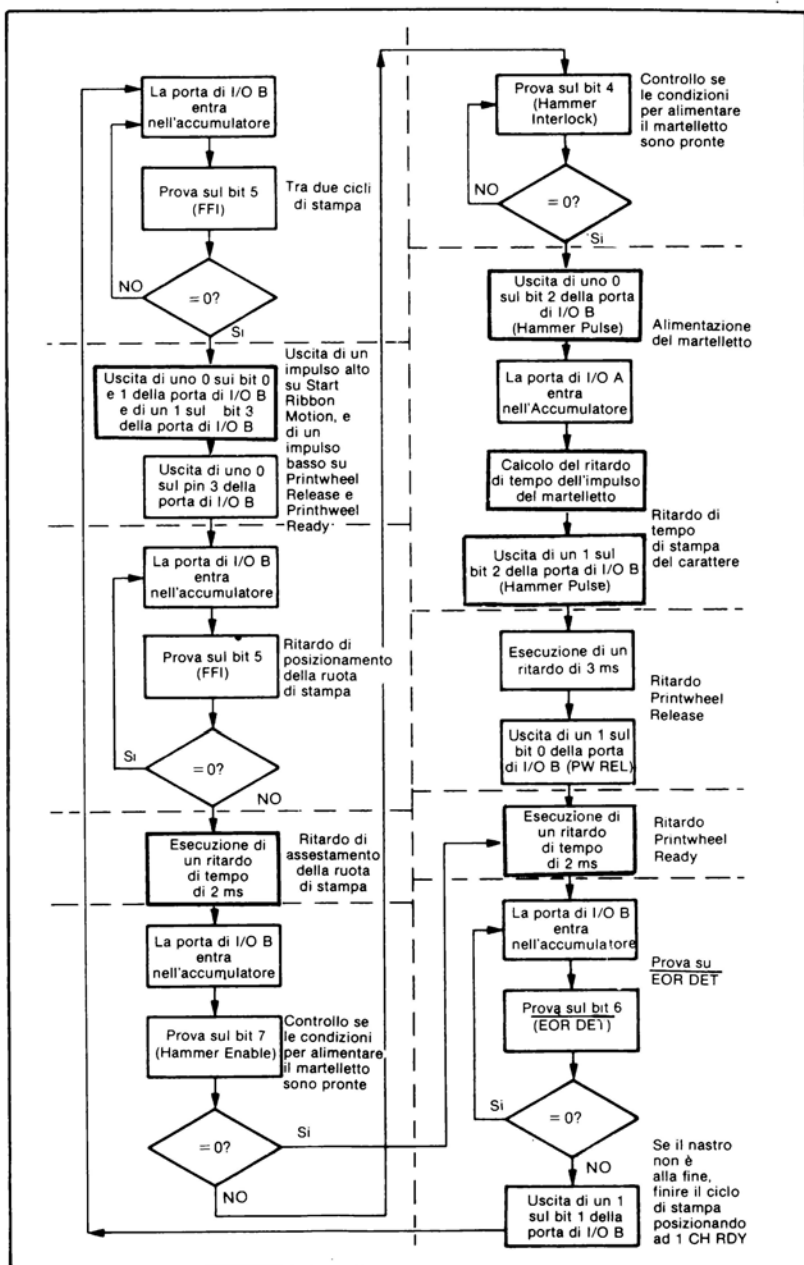



Figura 4-3. Primo tentativo di un flowchart del programma

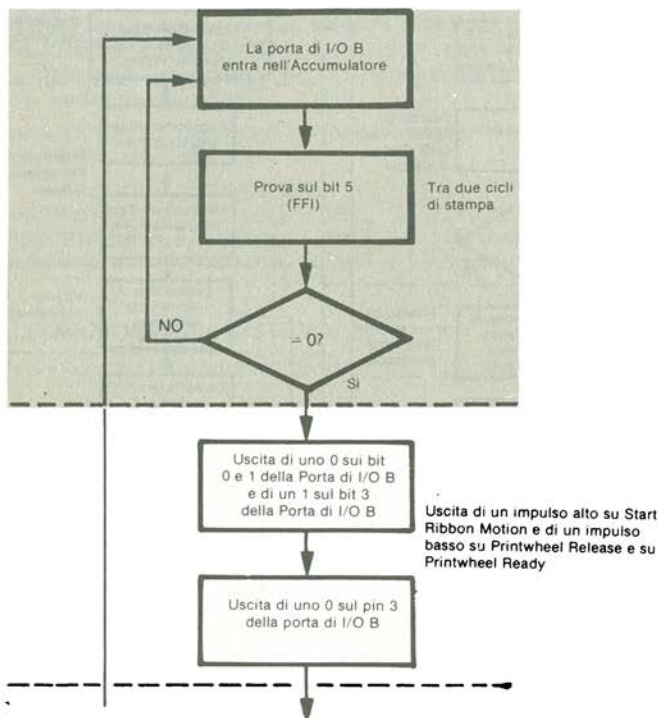
di stampa indietro nella sua posizione di visibilità? Il Reset abortirà un ciclo di stampa — perciò il programma del ciclo di stampa deve essere creato prima per sapere come abortirlo.

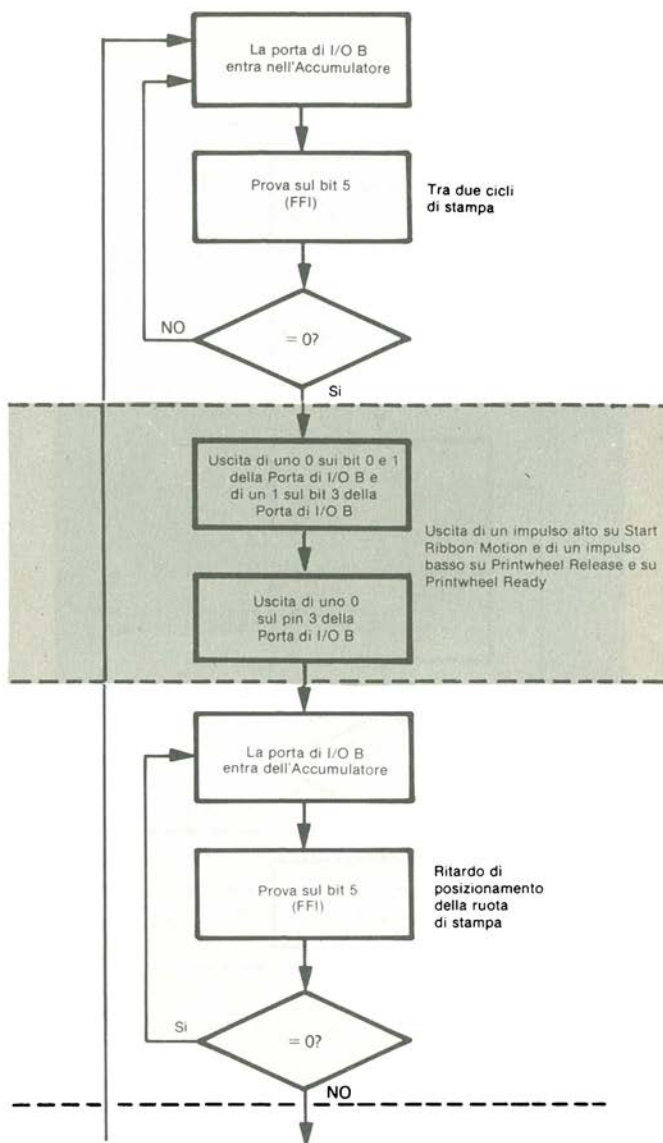
Generalmente voi potreste cominciare a scrivere un programma implementando l'evento più importante nella vostra logica, quindi potreste lavorare lontano da questo inizio e implementare eventi dipendenti.

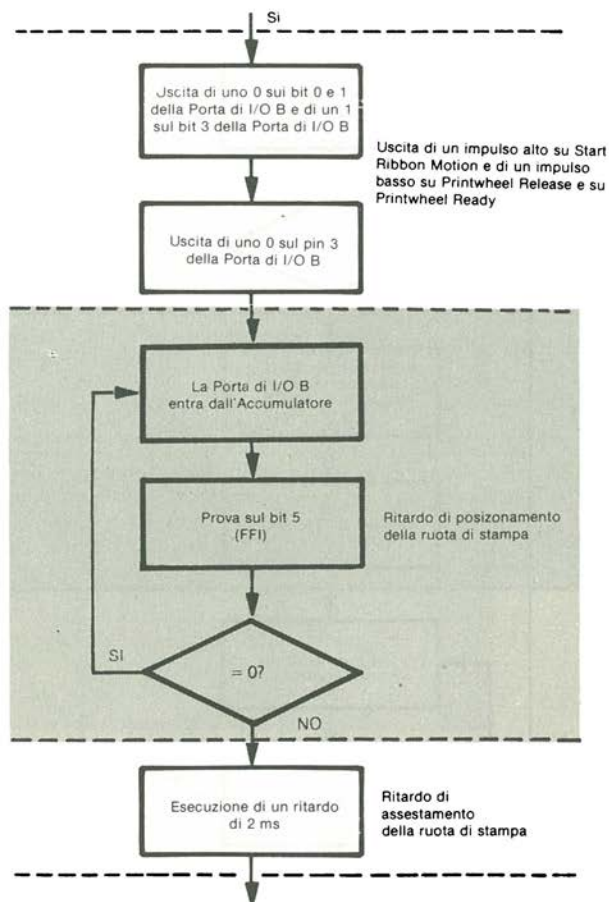
Intendiamo posporre la creazione di un programma per implementare la logica di Reset fino a che non si sia creato un programma di ciclo di stampa.

FLOWCHART DEL PROGRAMMA

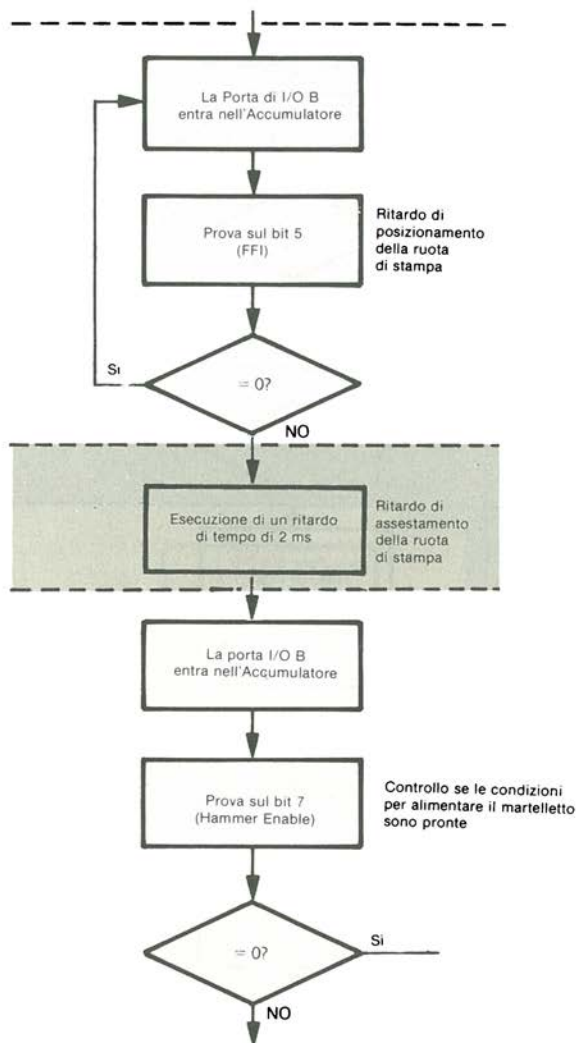
Volgiamo ora la nostra attenzione alle funzioni che devono essere realizzate dal sistema del microcalcolatore. Queste funzioni sono identificate dal flowchart illustrato in Figura 4-3. Analizzeremo questo flowchart a passo a passo.

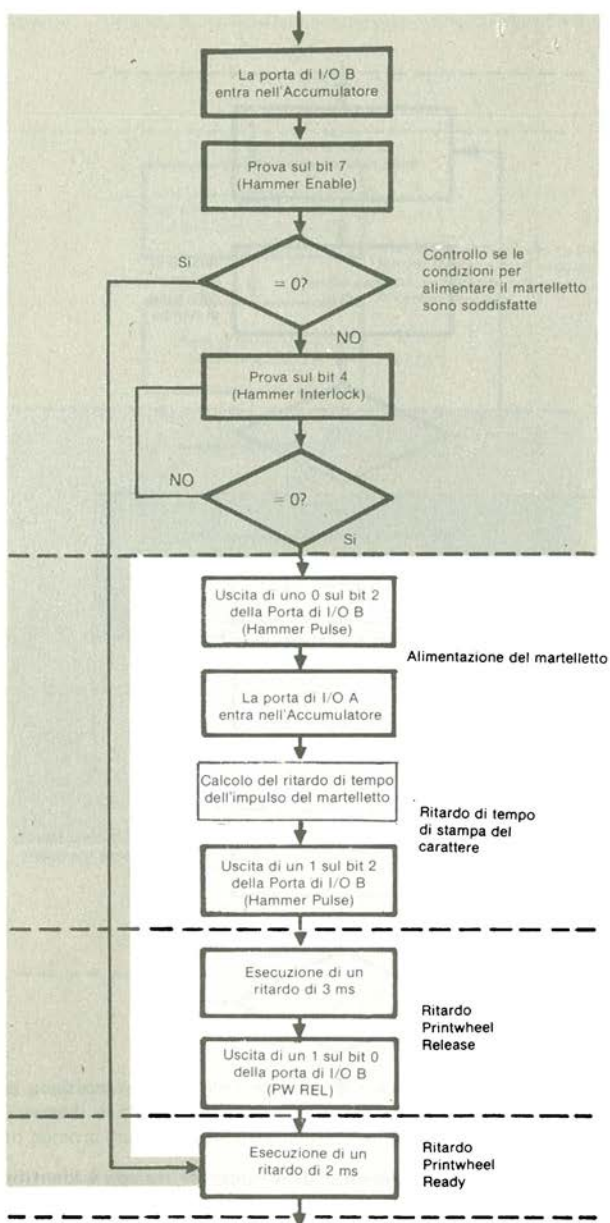
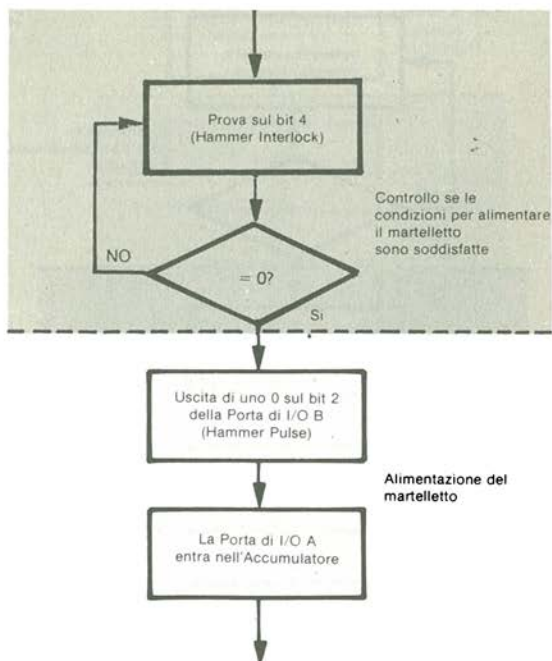
Useremo il segnale d'ingresso di decodifica della velocità (FF) per identificare l'inizio di un nuovo ciclo di stampa. Tra due cicli di stampa, perciò, il programma fa entrare continuamente il contenuto della Porta di I/O B nell'Accumulatore, facendo un test sul bit 5. Dal momento che questo bit è uguale ad 1, non è ancora cominciato un nuovo ciclo di stampa. Non appena questo è uguale a 0, si identifica un nuovo ciclo di stampa:

La prima cosa che accade in un nuovo ciclo di stampa è **l'uscita di un impulso alto sul START RIBBON MOTION scrivendo sequenzialmente un 1, poi uno 0 sul bit 3**

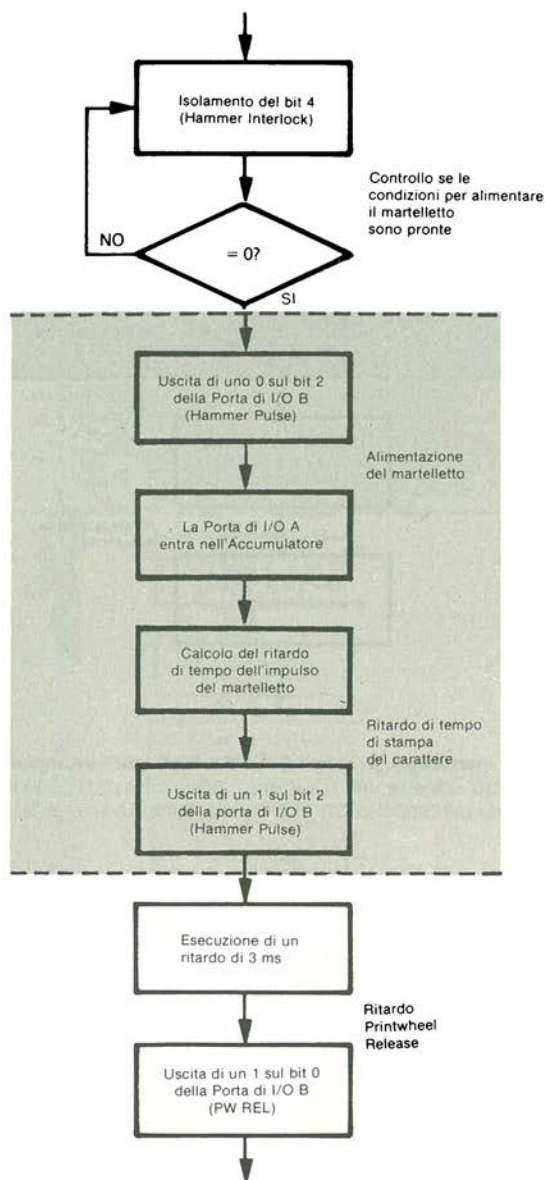
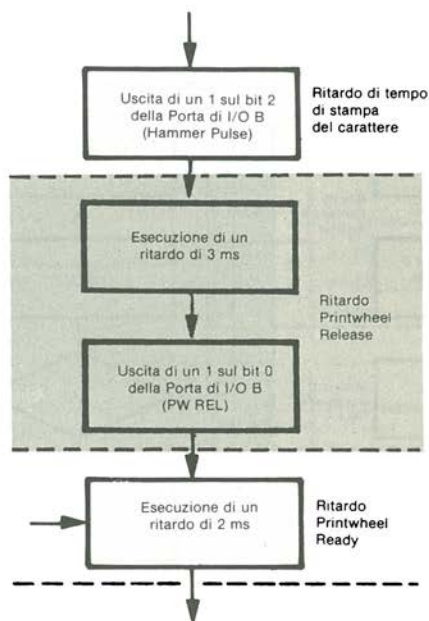
della Porta di I/O B. Inoltre si mettono degli zero sui bit 0 ed 1 della Porta di I/O B, poichè PRINTWHEEL RELEASE e PRINTWHEEL READY devono essere entrambi posizionati a 0 all'inizio del ciclo:

Il ritardo di posizionamento della ruota di stampa è calcolato dal segnale di decodifica della velocità FFI. Dal momento che questo segnale è basso, la ruota di stampa

si deve ancora posizionare. Andiamo perciò in un loop di ritardo variabile, che in termini di logica di programma, è l'inverso del loop di ritardo "tra due cicli di stampa". Ancora una volta, il contenuto della Porta di I/O B entra nell'Accumulatore e si controlla il bit 5; tuttavia rimaniamo nel loop di ritardo finchè il bit 5 non è 1. In questo istante il ritardo di posizionamento della ruota di stampa è completato:

Il ritardo di posizionamento della ruota di stampa deve essere seguito da un ritardo di due millisecondi di stabilizzazione della ruota di stampa. L'usuale loop di ritardo sarà eseguito come si può vedere nella figura di pagina 4-13.

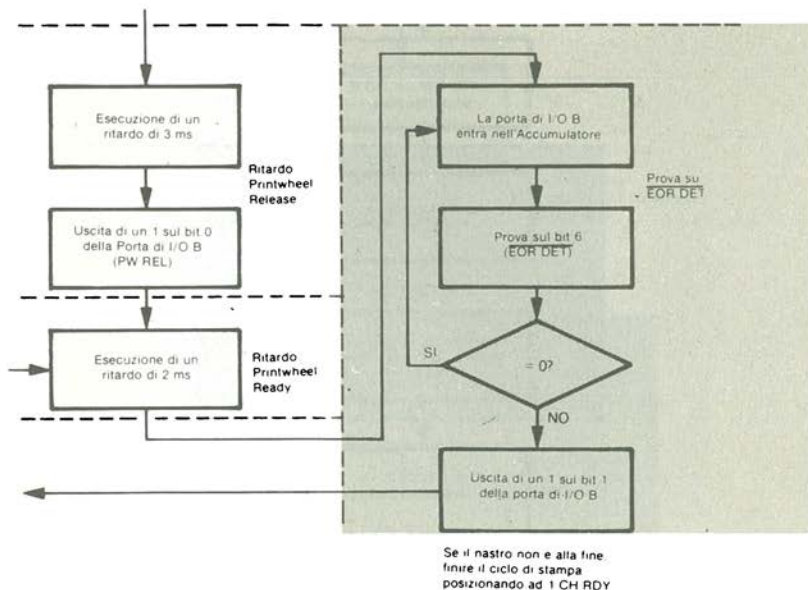
Alla fine del ritardo di stabilizzazione della ruota di stampa, **si alimenta il martelletto di stampa, stabilendo che il segnale HAMMER INTERLOCK sia basso e che HAMMER ENABLE sia alto.** Ricordiamo che HAMMER INTERLOCK è un bit di

stato di segnale, usato da tutte le condizioni esterne che possono impedire l'alimentazione del martelletto. Ogni segnale che mette un alto su questo stato sopprimerà l'alimentazione del martelletto di stampa.

Un ciclo di stampa di riposizionamento della ruota di stampa è identificato da un HAMMER ENABLE basso. Questa condizione è rilevata facendo un test sul bit 7 della Porta di I/O B prima di verificare la condizione di HAMMER INTERLOCK.

Se il bit 7 della Porta di I/O B è 0, allora si salta l'intera sequenza di alimentazione del martelletto di stampa e noi saltiamo direttamente al ritardo della ruota di stampa pronta, che è l'ultimo ritardo di tempo del ciclo di stampa:

Se HAMMER ENABLE è alto, questo è un ciclo di stampa di un carattere, così si alimenterà il martelletto di stampa, ma solo quando HAMMER INTERLOCK è 0. Dal momento che uno dei segnali in OR cablato sul pin 4 della Porta di I/O B è alto, il programma rimarrà in un loop senza fine, verificando continuamente lo stato di questo pin della Porta di I/O. Quando finalmente il pin della Porta di I/O è 0, il programma avanzerà alla sequenza di istruzioni che alimentano il martelletto di stampa:

Per alimentare il martelletto di stampa, si deve fare uscire un impulso di lunghezza variabile. Per far ciò si mette uno 0 sul pin 2 della Porta di I/O B, poichè questo pin è il mezzo di uscita dell'impulso del martelletto. Successivamente si calcola il ritardo

di tempo dell'impulso del martelletto. Descriveremo come si calcola la durata dello impulso del martelletto dopo aver completato la descrizione del flowchart. Alla fine del ritardo di tempo che alimenta il martelletto di stampa, si mette un 1 sul pin 2 della Porta di I/O B. Ciò fa terminare l'impulso che alimenta il martelletto di stampa:

Ora seguono due ritardi di stabilizzazione. Dapprima c'è un ritardo di tre millisecondi per lo sgancio della ruota di stampa, la fine del quale è indicata dall'uscita di un 1 sul bit 0 della Porta di I/O B. Ciò provoca l'uscita di un alto su PW REL:

Successivamente si esegue un ritardo di due millisecondi per rendere pronta la ruota di stampa. La fine di questo ritardo e **la fine del ciclo di stampa sono indicati da un 1 sul bit 1 della Porta di I/O B;** ciò posiziona CH RDY alto. **Noi, tuttavia, non vogliamo fare ciò, se c'è uno stato di fine nastro.** Questo stato è identificato da un basso su $\overline{\text{EOR DET}}$.

Il programma fa quindi entrare la Porta di I/O B nell'Accumulatore e fa un test sul bit 6, per mezzo del quale $\overline{\text{EOR DET}}$ entra nel sistema del microcalcolatore. Se $\overline{\text{EOR DET}}$ è 0, allora il programma rimane in un loop senza fine verificando conti-

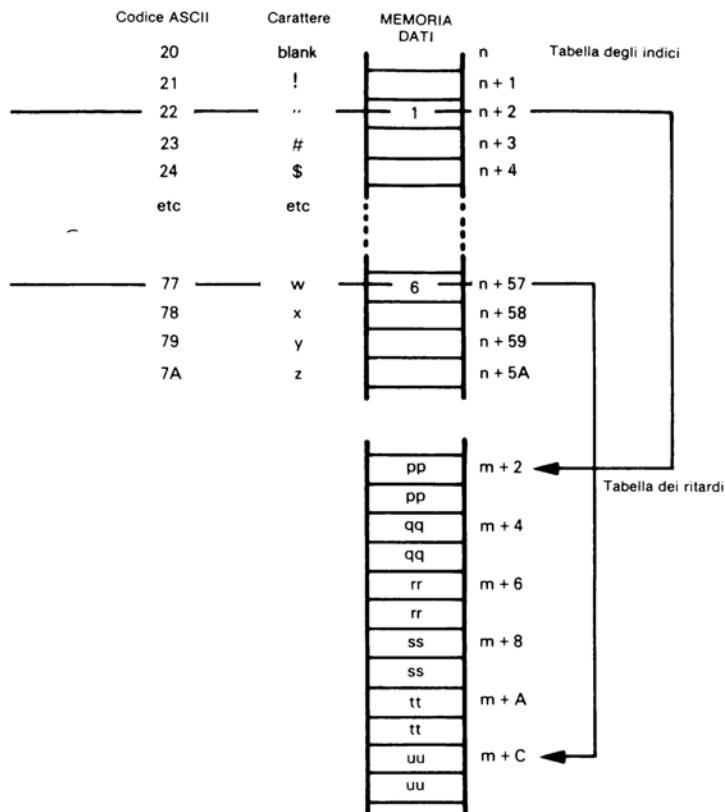
nuamente il bit 6 della Porta di I/O B; in tale modo un altro ciclo di stampa non può cominciare. Solo se si rileva che $\overline{\text{EOR DET}}$ è 1, il ciclo di stampa terminerà mettendo a 1 CH RDY:

RITARDO DELL'ALIMENTAZIONE DEL MARTELLETTO DI STAMPA

Volgiamo ora la nostra attenzione al metodo col quale si calcola l'appropriato ritardo per alimentare il martelletto di stampa. In Figura 3-1, l'appropriato ritardo per alimentare il martelletto di stampa era segnalato da una di sei linee (da H1 ad H6) che andava alta. Una logica esterna deve generare l'alto sulla linea, basandosi sulla natura del carattere che si sta stampando; **questo genere di operazione è più facile in un programma di un microcalcolatore.**

Questo è il metodo che useremo per calcolare l'appropriato ritardo di tempo che alimenta il martelletto di stampa: ogni carattere da stampare è rappresentato da un solo byte di dato in codice ASCII, come illustrato nell'Appendice A.

Se ignoriamo il bit di parità (bit di ordine maggiore), allora rimangono 128 possibili combinazioni. Se guardate i codici ASCII dati nell'Appendice A, vedrete che sono significativi solo i codici di caratteri tra 20_{16} e $7A_{16}$. Perciò bisogna render conto di sole $5A_{16}$ (o 90_{10}) combinazioni di codici. Ognuna di queste combinazioni di codici avrà un solo byte in una tabella di 90 byte; in questi byte sarà memorizzato un numero tra 1 e 6.

Questo numero identificherà il ritardo di tempo richiesto dal carattere. Una tabella di 12 byte conterrà i sei ritardi di tempo reali associati ai sei bit. Questo schema

può essere illustrato come segue:

Nell'illustrazione precedente le lettere "n" e "m", alla destra della memoria dei dati, rappresentano ogni valido indirizzo di base della memoria. Per esempio, "n" può rappresentare 0390_{16} mentre "m" rappresenta $03F0_{16}$.

Consideriamo due esempi

Il codice ASCII 22_{16} indica il carattere doppio indice ("), che richiede il ritardo di tempo più breve. Il byte di memoria dati con indirizzo n+2 corrisponde a questo codice ASCII. Un 1 è memorizzato in questo byte di memoria dati. Perciò, il primo ritardo di tempo, rappresentato da pppp, è il valore che deve essere caricato nel registro Indice prima di eseguire il loop di ritardo di tempo breve che crea l'impulso che fa sparare il martelletto di stampa per il carattere".

Il codice ASCII 77_{16} rappresenta "w". Il byte di memoria dati con indirizzo $n+57_{16}$ corrisponde a questo codice ASCII. In questo byte di memoria dati è memorizzato il valore 6, che indica che per "w" è richiesto il ritardo di tempo più lungo per alimentare il martelletto di stampa. Perciò, si caricherà un valore rappresentato da uuuu nel registro Indice, prima di eseguire il loop di ritardo di tempo lungo che crea l'impulso che alimenta il martelletto di stampa per il carattere w.

La Figura 4-4 identifica i passi del programma col quale si calcolerà il ritardo che fa sparare il martelletto di stampa.

Ⓐ Ingresso del codice ASCII del carattere dalla Porta A

Ⓑ Posizionamento a 0 del bit 7

Ⓒ Sottrazione di 20_{16}

Ⓓ Somma il risultato all'indirizzo di base della Tabella degli Indici

Ⓔ Ingresso dell'indice

Ⓕ Moltiplicazione per 2

Ⓖ Somma all'indirizzo base della Tabella dei Ritardi

Ⓗ Il ritardo entra nei registri DE

Ⓘ Esecuzione del ritardo che alimenta il martelletto

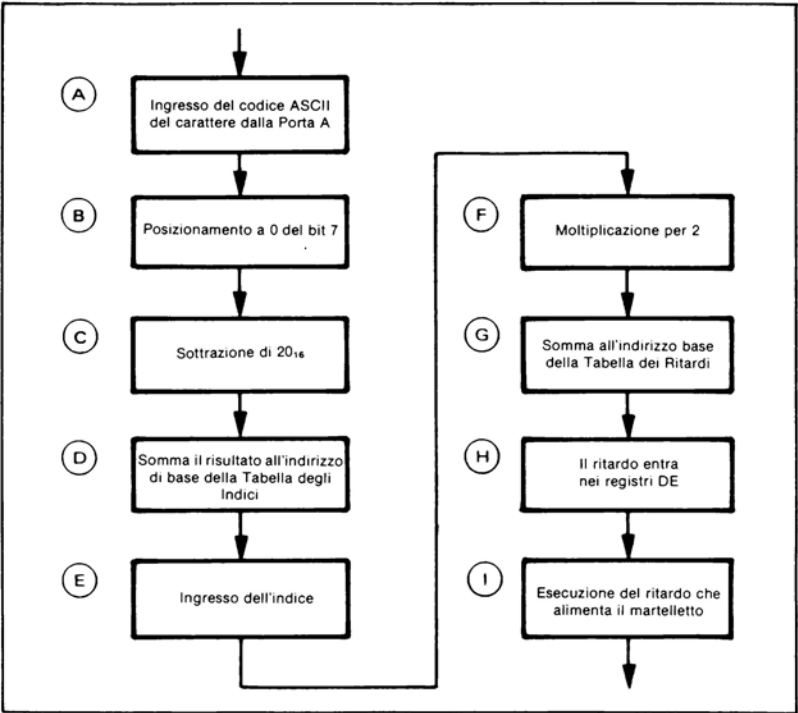



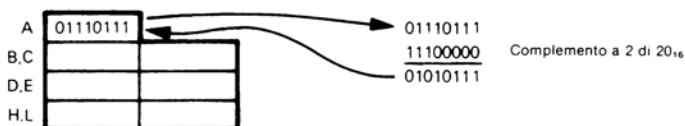
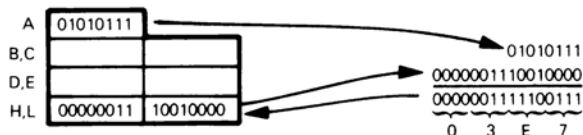
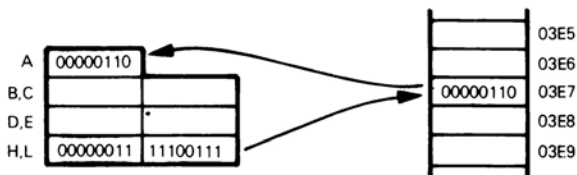
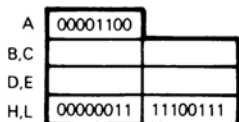
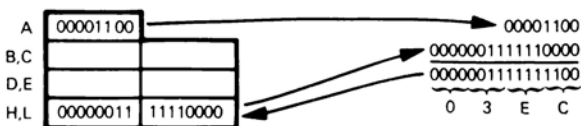
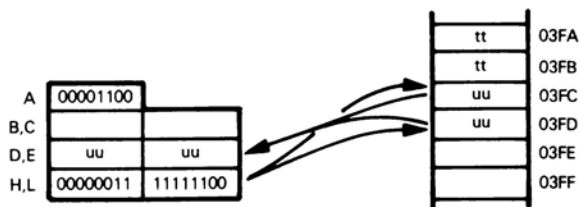
Figura 4-4. Flowchart del programma per calcolare la lunghezza dell'impulso che alimenta il martelletto di stampa

Per capire meglio la Figura 4-4, andremo dal passo Ⓐ fino al passo Ⓘ per il caso di "w".

Ⓐ La rappresentazione ASCII della lettera minuscola w entra nell'Accumulatore:

A	X1110111	← Dalla porta di I/O A
B,C		
D,E		
H,L		

Ⓑ Posizioniamo poi a 0 il bit di parità;

Ⓒ L'ingresso nella tabella indice corrispondente alla lettera minuscola w è calcolato sommando il codice ASCII meno 20_{16} all'indirizzo di base della tabella indice. Dobbiamo sottrarre 20_{16}, perchè i primi $1F_{16}$ codici non hanno un equivalente in ASCII:

Ⓓ L'indirizzo base della tabella indice è caricato nei registri H ed L. Supporremo che questo indirizzo sia 0390_{16}. Allora il contenuto dell'Accumulatore è sommato all'indirizzo di 16 bit:

Ⓔ L'indirizzo appropriato è caricato nell'Accumulatore della tabella indice:

Ⓕ Poichè il ritardo reale è lungo due byte, calcoleremo l'indirizzo del ritardo appropriato sommando due volte l'indice all'indirizzo base della tabella indice. Moltiplichiamo dapprima l'indice per 2:

Ⓖ Successivamente sommiamo l'indice moltiplicato per 2 all'indirizzo base della tabella dei ritardi. Supponiamo che questo indirizzo base sia $03F0_{16}$. Questo indirizzo base è nuovamente caricato nei registri H ed L, dopo di chè il contenuto dell'Accumulatore è sommato al contenuto dei registri H ed L:

(H) I due byte indirizzati da H e L sono caricati nei registri D e F:

(I) I registri D ed E ora contengono il valore iniziale corretto per un ritardo lungo che è eseguito come descritto nel Capitolo 2.

```
; Programma del ciclo di stampa
; Tra due cicli di stampa verifica su FF1 (bit 5 della Porta di I/O B) per un valore zero
START:  IN    A,(2)         ; La Porta di I/O B entra nell'Accumulatore
        BIT   5,A           ; Test sul bit 5
        JR    NZ,START      ; Se non è zero, torna a START
; Inizializzazione del ciclo di stampa. Posiziona 0 sui bit 0 ed 1 della Porta di I/O B,
; posiziona 1 sui bit 2 e 3 della Porta di I/O B
        LD    A,0CH         ; Carica la maschera nell'Accumulatore
        OUT   (2),A         ; Uscita verso la Porta di I/O B
; Posiziona a 0 del bit 3 della Porta di I/O B, completando l'impulso START RIBBON
; MOTION
        RES   3,A           ; Azzeramento del bit 3 della maschera nell'Accu-
                            ; mulatore
        OUT   (2),A         ; Uscita della Porta di I/O B
; Test sulla fine del posizionamento della ruota di stampa
; Il bit 5 della Porta di I/O B (FFI) sarà 1
PWPOS:  IN    A,(2)         ; La Porta di I/O B entra nell'Accumulatore
        BIT   5,A           ; Test sul bit 5
        JR    Z,PWPOS       ; Se è 0, tornare ancora al test
; Esecuzione del ritardo di 2 millisecondi di stabilizzazione della ruota di stampa
        LD    A,0FAH        ; Carica la costante iniziale del ritardo di tempo
PWSET:  DEC   A             ; Decremento dell'Accumulatore
        JR    NZ,PWSET      ; Ridecremento dell'Accumulatore se non è zero
; Test sulle condizioni per l'alimentazione del martelletto
PHFIR:  IN    A,(2)         ; La Porta di I/O B entra nell'Accumulatore
        BIT   7,A           ; Test sul bit 7 (HAMMER ENABLE)
        JP    Z,PWRDY       ; Se è 0, superare l'alimentazione del martelletto
        BIT   4,A           ; Test su HAMMER INTERLOCK
        JR    Z,PHFIR       ; Attesa per un valore diverso da zero prima della
                            ; alimentazione
; Alimentazione del martelletto
        RES   2,A           ; Posizionamento a basso di HAMMER PULSE
        OUT   (2),A         ; Uscita di uno 0 sul bit 2 della Porta di I/O B
        IN    A,(0)         ; Il carattere ASCII entra nell'Accumulatore
        RES   7,A           ; Azzeramento del bit di ordine maggiore
        SUB   20H           ; Sottrazione di 20H
```

Figura 4-5. Una semplice sequenza di istruzioni di un ciclo di stampa senza inizializzazione o reset (segue)

```
          LD    HL,INDX         ; Carica l'indirizzo base della tabella degli indici
                                ; HL
          ADD   L               ; Somma il contenuto dell'Accumulatore ad HL
          LD    L,A
          LD    A,(HL)          ; Carica l'indice nell'Accumulatore
          ADD   A               ; Moltiplicazione per 2
          LD    HL,DELY         ; Caricamento dell'indirizzo base della tabella dei
                                ; ritardi in HL
          ADD   L               ; Somma del contenuto dell'Accumulatore ad HL
          LD    L,A
          LD    E,(HL)          ; Caricamento della costante di tempo in D, E
          INC   HL
          LD    D,(HL)
PRDLY:    DEC   DE              ; Esecuzione del ritardo di stampa
          LD    A,D
          OR    E
          JR    NZ,PRDLY
          IN    A,(2)           ; Alla fine del ritardo esce un 1 sul bit 2 della Porta
          SET   2,A             ; di I/O B. Ciò posiziona alto HAMMER PULSE
          OUT   (2),A
; Esecuzione di un ritardo di tempo di 3 millisecondi per sganciare la ruota di stampa
          LD    DE,231          ; Caricamento della costante iniziale del ritardo di
                                ; tempo
PWREL:    DEC   DE              ; Esecuzione del ritardo di tempo lungo
          LD    A,D
          OR    E
          JR    NZ,PWREL
; Uscita di un 1 sul bit 0 della Porta di I/O B. Ciò posiziona alto PW REL
          IN    A,(2)           ; La Porta di I/O B entra nell'Accumulatore
          SET   0,A             ; Posizionamento a 1 del bit 0
          OUT   (2),A           ; Uscita del risultato
; Esecuzione di un ritardo di tempo di 2 millisecondi per rendere pronta la ruota di
; stampa
PWRDY     LD    A,0FAH          ; Carica la costante iniziale del ritardo di tempo
RDYDLY:   DEC   A               ; Decremento dell'Accumulatore
          JR    NZ,RDYDLY       ; Ridecremento se non è zero
. Test per EOR DET uguale a 1 (bit 6 della Porta di I/O B) come prerequisito per
; finire il ciclo di stampa
EORCHK:   IN    A,(2)           ; La Porta di I/O B entra nell'Accumulatore
          BIT   6,A             ; Test sul bit 6
          JR    Z,EORCHK        ; Se è 0 ritorna e ripeti il test
, Alla fine del ciclo di stampa posizionare a 1 il bit 1 della Porta di I/O B ciò posizio-
; na alto CH RDY
          SET   1,A             ; Posizionamento a 1 del bit 1 della Porta B (nel-
                                ; l'Accumulatore)
          OUT   (2),A           ; Uscita del risultato
          JP    START           ; Salto al test del nuovo ciclo di stampa
```

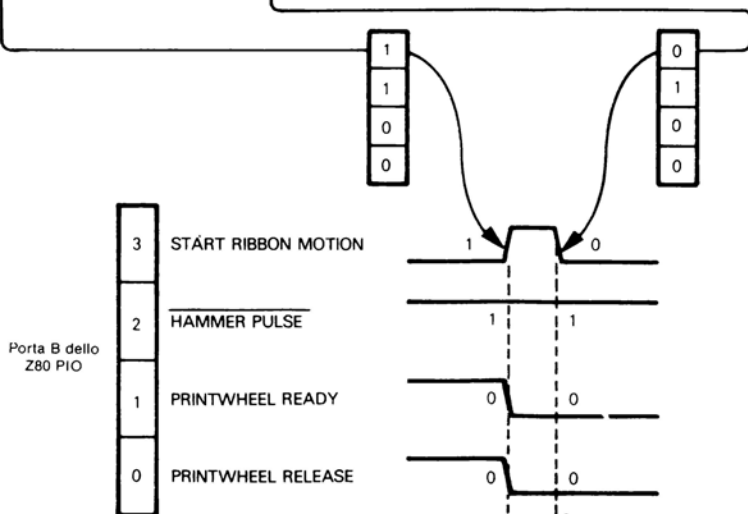
Figura 4-5. Una semplice sequenza di istruzioni di un ciclo di stampa senza inizializzazione o reset

Mettendo insieme i flowcharts dei programmi illustrati nelle Figure 4-3 e 4-4, si genera l'intero programma richiesto, come illustrato in Figura 4-5. Descriviamo questo programma, sezione per sezione.

Tra due cicli di stampa, il seguente loop di tre istruzioni verifica continuamente lo stato della Porta di I/O B, bit 5. Il segnale FFI è messo in ingresso su questo pin. Dal momento che questo segnale d'ingresso è alto, non può cominciare nessun nuovo ciclo di stampa. Non appena questo segnale ha l'ingresso basso, la ruota di stampa viene identificata come se fosse in movimento — che significa che è in via di esecuzione un nuovo ciclo di stampa:

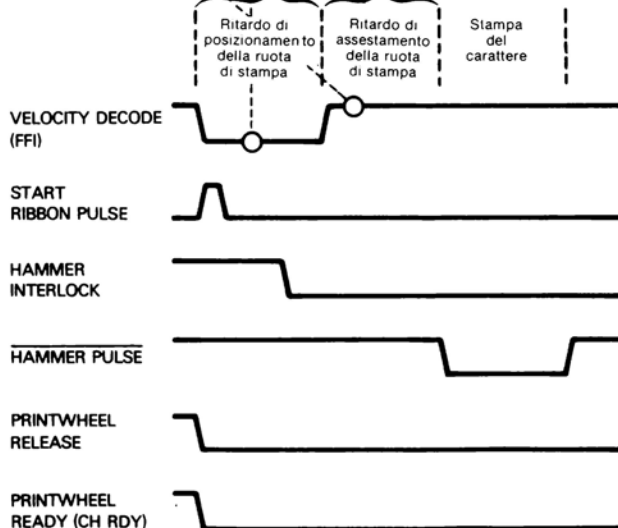
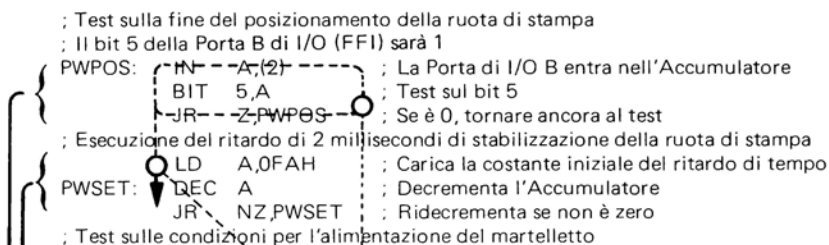
```
; Programma del ciclo di stampa
; Tra due cicli di stampa verifica su FFI (bit 5 della Porta di I/O B) per un valore zero
Ingresso        START: IN   A,(2)       ; La Porta di I/O B entra nell'Ac-
nel Programma                           ; cumulatore
                       BIT  5,A         ; Test sul bit 5
                       JR   NZ,START    ; Se non è zero, tornare a START
; Inizializzazione del ciclo di stampa. Uscita di uno 0 sui bit 0 ed 1 della Porta di I/O
; B, uscita di un 1 sui bit 2 e 3 della Porta di I/O B
                       LD   A,0CH       ; Carica la maschera nell'Accumu-
                                        ; latore
```

Non appena comincia un nuovo ciclo di stampa, i segnali PRINTWHEEL RELEASE e PRINTWHEEL READY devono essere messi bassi. Inoltre si deve far uscire un impulso di alto su START RIBBON MOTION cosicchè, quando si alimenta il martelletto di stampa, un nastro fresco è di fronte al carattere che deve essere stampato. Questi cambiamenti iniziali dei segnali possono essere illustrati come segue:

```
; Inizializzazione del ciclo di stampa. Uscita di uno 0 sui bit 0 ed 1 della Porta di I/O
; B, uscita di un 1 sui bit 2 e 3 della Porta di I/O B
         LD    A,0CH       ; Carica la maschera nell'Accumulatore
         OUT   (2),A       ; Uscita verso la Porta di I/O B
; Uscita di uno 0 sul bit 3 della Porta di I/O B, completando l'impulso START
; RIBBON MOTION
         RES   3,A         ; Azzeramento del bit 3 della maschera nell'Ac-
                           ; cumulatore
         OUT   (2),A       ; Uscita verso la Porta di I/O B
```

IMPULSO DI SEGNALE PROGRAMMATO

Nella illustrazione precedente è da notare che il pin 2 della Porta di I/O B è stato forzato ad 1 in uscita. Questo è il pin $\overline{\text{HAMMER PULSE}}$, che va basso solo per la durata dell'impulso che alimenta il martelletto di stampa. A questo punto del ciclo di stampa questo segnale è alto, cosicchè l'uscita di un 1 è innocua.

RITARDO DI TEMPO DI LUNGHEZZA VARIABILE

Ora il programma esegue un ritardo di lunghezza variabile, durante il quale la ruota di stampa si muove finchè il petalo del carattere appropriato sia di fronte al martelletto di stampa, o finchè la ruota di stampa ritorni nella sua posizione di visibilità. Nell'uno o nell'altro caso, la logica esterna posiziona basso il segnale di ingresso FFI per la durata del ritardo del posizionamento della ruota di stampa. Non appena la ruota di stampa si è posizionata, si rileva FFI alto — e **la logica del programma avanza al ritardo di due millisecondi del posizionamento della ruota di stampa.** Prima abbiamo visto frequentemente questo loop di ritardo di tre istruzioni:

Ora il martelletto di stampa è pronto per essere alimentato. Dapprima facciamo un test sulla condizione di HAMMER ENABLE, che è stato collegato al pin 7 della Porta di I/O B. Se questo segnale è basso, allora noi siamo in un ciclo di stampa di riposizionamento della ruota di stampa e si salta la sequenza di istruzioni che alimentano il martelletto. **Se HAMMER ENABLE è alto, noi superiamo questo test. Ma si deve ancora verificare HAMMER INTERLOCK; questo segnale è in ingresso sul pin 4 della Porta di I/O B.** Poichè l'istruzione BIT che fa il test sul bit 7 lascia intatto il contenuto dell'Accumulatore, eseguiremo semplicemente un'altra istruzione BIT per verificare HAMMER INTERLOCK.

Se si rileva HAMMER ENABLE basso, l'esecuzione salta alla istruzione di etichetta PWRDY. Troverete questa istruzione vicina alla fine del programma, all'inizio della sequenza di istruzioni che esegue **un ritardo di due millisecondi PRINTWHEEL READY.**

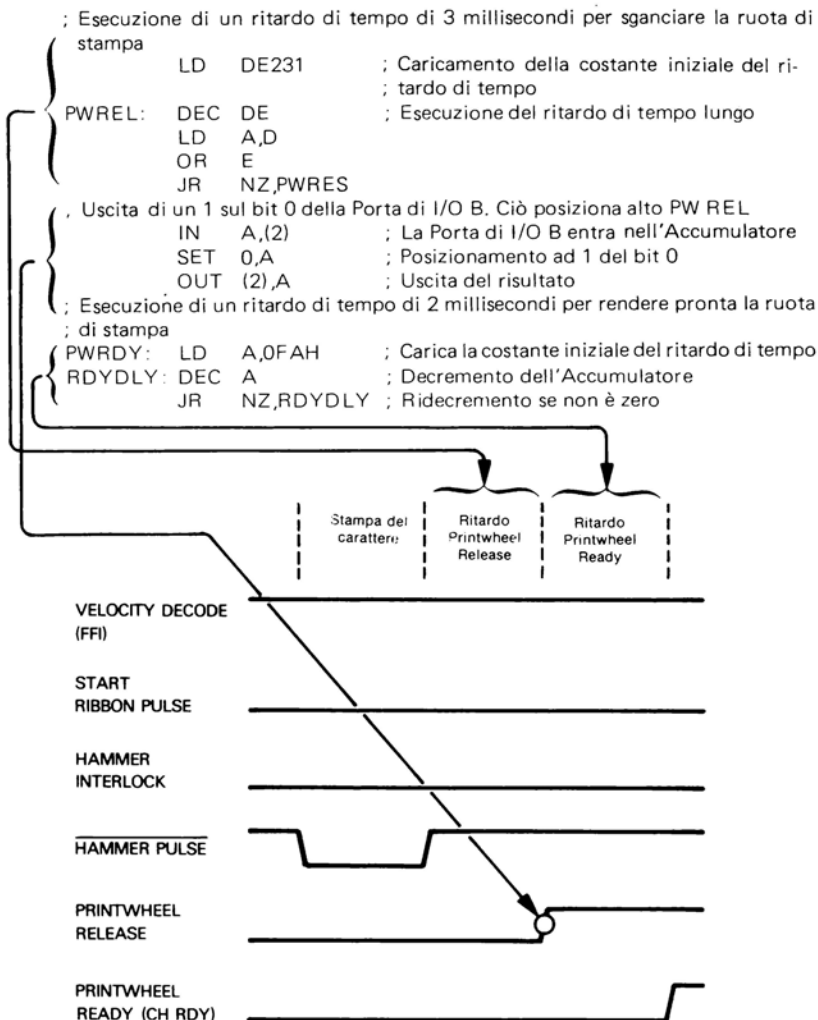
Notate che la sequenza di cinque istruzioni illustrata in Figura 4-5 fa il test su HAMMER ENABLE basso nel loop che fa il test su HAMMER INTERLOCK alto. HAMMER ENABLE sarà alto o basso per la durata del ciclo di stampa; non cambierà livello durante il ciclo di stampa. Perciò, il fatto che esso sia continuamente verificato è ridondante — non serve a nessun scopo ma non dà noie.

Successivamente si alimenta il martelletto di stampa. La sequenza di istruzioni che provoca l'alimentazione del martelletto di stampa implementa i passi da Ⓐ a Ⓘ, che abbiamo già descritto. Per rendere la sequenza di istruzioni più facile da comprendere, essa è riprodotta qui sotto con aggiunte le etichette da Ⓐ a Ⓘ :

```
; Alimentazione del martelletto di stampa
                RES   2,A        ; Posizionamento a basso di HAMMER PULSE
                OUT   (2),A      ; Uscita di uno 0 sul bit 2 della Porta di I/O B
      Ⓐ         IN    A,(0)      ; Il carattere ASCII entra nell'Accumulatore
      Ⓑ         RES   7,A        ; Azzeramento del bit di ordine maggiore
      Ⓒ         SUB   20H        ; Sottrazione di 20H
              { LD    HL,INDX    ; Carica l'indirizzo base della tabella degli indici in
      Ⓓ       {                  ; HL
              { ADD   L          ; Somma il contenuto dell'Accumulatore ad HL
              { LD    L,A
      Ⓔ         LD    A,(HL)     ; Carica l'indice nell'Accumulatore
      Ⓕ         ADD   A          ; Moltiplica per 2
              { LD    HL,DELY    ; Carica l'indirizzo base della tabella dei ritardi in
      Ⓖ       {                  ; HL
              { ADD   L          ; Somma il contenuto dell'Accumulatore ad HL
              { LD    L,A
              { LD    E,(HL)     ; Carica la costante di ritardo in D, E
      Ⓗ       { INC   H,L
              { LD    D,(HL)
PRDLY         { DEC   DE         ; Esecuzione del ritardo di stampa
      Ⓘ       { LD    A,D
              { OR    E
              { JR    NZ,PRDLY
                IN    A,(2)      ; Alla fine del ritardo esce un 1 sul bit 2 della
                SET   2,A        ; porta di I/O B. Ciò posiziona alto HAMMER
                                 ; PULSE
                OUT   (2),A
; Esecuzione del ritardo di tempo di 3 millisecondi PRINTWHEEL RELEASE
```

Notate che i test sui bit di HAMMER ENABLE e di HAMMER INTERLOCK hanno lasciato intatto il contenuto della Porta B nell'Accumulatore. Non abbiamo bisogno

quindi di nessun ingresso dalla Porta B, prima di posizionare basso $\overline{\text{HAMMER PULSE}}$; posizioniamo semplicemente a 0 il bit 2 dell'Accumulatore e quindi mettiamo il risultato nella Porta di I/O B.

Si esegue ora un ritardo di tempo di tre millisecondi PRINTWHEEL RELEASE, e la fine di questo ritardo di tempo è indicato da un impulso alto sul segnale PRINTWHEEL RELEASE. Successivamente si esegue il ritardo di due millisecondi PRINTWHEEL READY.

Prima di terminare il ciclo di stampa facendo uscire un impulso alto su PRINTWHEEL READY (CH RDY), il programma si deve assicurare che non si è raggiunta la fine del

nastro. Se si rileva $\overline{\text{EOR DET}}$ basso, il programma rimane in un loop senza fine finchè non si cambia il nastro; allora la logica esterna metterà alto $\overline{\text{EOR DET}}$. Quando si rileva $\overline{\text{EOR DET}}$ alto le istruzioni finali del programma posizionano alto PRINTWHEEL READY, poi ritornano all'inizio del programma e aspettano il nuovo ciclo di stampa.

ERRORI LOGICI DEL PROGRAMMA

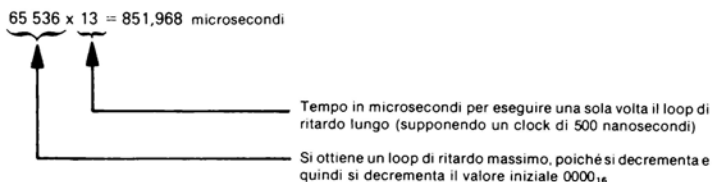
Il programma sviluppato in questo capitolo contiene un errore logico che potrebbe non verificarsi in una implementazione di logica digitale. L'errore è nel calcolo del ritardo di tempo dell'impulso del martelletto.

In una implementazione con logica digitale, il codice ASCII per ogni carattere sarebbe elaborato come sette segnali individuali. Questi segnali verrebbero combinati in qualche maniera per generare uno dei segnali di ritardo di tempo H1 ÷ H6. **Non importa quale combinazione di codice ASCII ci sia in ingresso, in uscita ci sarà uno dei segnali di ritardi di tempo da H1 ad H6;** se la logica di generazione del segnale non è buona, si creerà pure un segnale di ritardo di tempo, sebbene esso possa essere un segnale errato.

LIMITAZIONE DELLA RICERCA

Guardiamo ora la implementazione di un programma in linguaggio assembly. E' abbastanza semplice per noi guardare la tabella dell'Appendice A e vedere che i codici ASCII validi coprono solo il campo da 20_{16} a $7A_{16}$. Ciò non impedisce ad un progettista logico di usare il sistema a microcalcolatore in un sistema speciale che includa caratteri insoliti, rappresentati da codici al di fuori del normale campo ASCII. In queste condizioni il nostro programma potrebbe dare risultati molto strani. Supponiamo che si sia adottato il codice ASCII 10_{16} per rappresentare un carattere speciale. Allora il nostro tentativo di ricerca nella Tabella Indice caricherebbe nell'Accumulatore ciò che era nel byte di memoria $n-10_{16}$.

Non si può dire che cosa ci potrà essere in questo byte di memoria; con tutta probabilità questo byte sarà usato per memorizzare un codice di un'istruzione, forse un valore di due digit esadecimali. Supponiamo ch esso contenga $2A_{16}$; il prossimo passo del programma raddoppierà $2A_{16}$, addizionandolo all'indirizzo base della Tabella dei Ritardi, ed accede al codice del ritardo iniziale dalla locazione di memoria $m+54_{16}$.

Data la configurazione di microcalcolatore di Figura 4-2, questa locazione di memoria potrebbe essere facilmente uno degli indirizzi duplicati che accedono illegittimamente a qualche byte di memoria, poichè si è usata una logica di selezione del chip molto semplice. Se avessimo usato una logica di selezione del chip più complessa, ci sarebbero state delle possibilità che ora stessimo accedendo a un byte di memoria non esistente. Nel caso precedente, non si può dire niente sulla lunghezza dell'impulso del martelletto generato; nell'ultimo caso si genererebbe un impulso di martelletto estremamente lungo, poichè riporteremmo 0 da una locazione di memoria non esistente, e questo valore sarebbe interpretato come costante iniziale di ritardo per il loop di programma del ritardo lungo. L'impulso del martelletto sarebbe lungo 852 millisecondi:

Per evitare questo problema ci sono due possibilità:

1) **La logica del programma può semplicemente ignorare ogni codice ASCII non valido.**
2) **La logica del programma può generare un impulso di martelletto con durata in difetto per codici ASCII non validi.**

Se ignoriamo i caratteri speciali, la conclusione è ovvia: il sistema del microcalcolatore non può essere usato in un'applicazione che richiede la stampa di caratteri speciali. Poichè il carattere speciale è ignorato, non accadrà niente quando si rileva in ingresso un codice di tale carattere — non ci sarà nessun impulso del martelletto, nessun movimento del carrello e nessun posizionamento.

Il fornire un impulso di martelletto in difetto per caratteri speciali significa che tali caratteri non saranno stampati, ma che si possono creare disuguaglianze nella densità del testo stampato.

Come progettisti logici, dovrete specificare la vostra preferenza.

Nel programma esistente si può inserire l'una o l'altra sequenza di istruzioni, come segue:

```
                IN    A,(0)       ; Il carattere ASCII entra nell'Accumulatore
                RES   7,A         ; Azzera il bit di peso maggiore
            --> SUB   20H         ; Sottrai 20H
                LD    HL,INDX     ; Carica in HL l'indirizzo di base della tabella degli
Qui si in-                        ; indici
serisce il      ADD   L           ; Addiziona il contenuto dell'Accumulatore ad HL
test sui co-    LD    L,A
dici ASCII      LD    A,(HL)      ; Carica l'indice nell'Accumulatore
validi          ADD   A           ; Moltiplica per 2
```

Ecco la sequenza di istruzioni che ignora i codici ASCII non standard:

```
                •
                •
                •
                IN    A,(0)       ; Il carattere ASCII entra nell'Accumulatore
                RES   7,A         ; Azzera il bit di peso maggiore
; Confronta il codice ASCII col più basso valore legittimo
                CP    20H
                JP    M,PWRDY     ; Se il codice è 1FH o minore, superare l'alimenta-
                                  ; zione del martelletto
, Confronta il codice ASCII col più alto valore legittimo
                CP    7BH
                JP    P,PWRDY     ; Se il codice è 7BH o maggiore, superare l'alimen-
                                  ; tazione del martelletto
; Il codice ASCII è valido
                SUB   20H         ; Sottrai 20H
                •
                •
                •
```

La seconda opzione, illustrata sotto, stampa i caratteri non noti con una densità media, usando il codice di densità 3:

```
                •
                •
                •
                IN    A,(0)       ; Il carattere ASCII entra nell'Accumulatore
                RES   7,A         ; Azzera il bit di peso maggiore
```

```
, Confronta il codice ASCII col più basso codice legittimo
            CP    20H
            JP    P,OK         ; Se il codice è 20H o maggiore, fare il test per il
                               ; limite superiore
; Il codice è illegittimo. Supporre una densità 3
NOK:        LD    A,6          ; Carica la densità doppia
            JP    NEXT
; Confronta il codice ASCII col valore legittimo più grande
OK:         CP    7BH          ; Se il codice è 7BH o maggiore, supporre una
            JP    P,NOK        ; una densità 3
; Il codice ASCII è valido
            SUB   20H          ; Sottrai 20H
            LD    HL,INDX      ; Carica in HL l'indirizzo di base della tabella degli
                               ; indici
            ADD   L            ; Addiziona il contenuto dell'Accumulatore ad HL
            LD    L,A
            LD    A,(HL)       ; Carica l'indice nell'Accumulatore
            ADD   A            ; Moltiplica per 2
NEXT:       LD    HL,DELY      ; Carica l'indirizzo base della tabella dei ritardi in
                               ; HL
            •
            •
            •
```

Entrambi le sequenze di istruzioni dei codici ASCII non validi sono semplicistiche nella loro soluzione del problema.

CONFRONTO IMMEDIATO

SALTO SU CONDIZIONE

L'unico aspetto nuovo introdotto è l'uso dell'istruzione Compare Immediate (CP). Questa istruzione sottrae i dati immediati nell'operando dal contenuto dell'Accumulatore. Il risultato della sottrazione è scartato, il che significa che il contenuto dell'Accumulatore non è alternato; tuttavia, si posizionano i flag degli stati per rispecchiare i risultati della sottrazione. Si usa un'istruzione JP M (Jump on Minus) per identificare un risultato negativo, il che significa che il dato immediato nell'operando era più grande del valore dell'Accumulatore. Analogamente, una istruzione JP P (Jump on Plus) identifica un valore immediato nell'operando che è uguale o minore del contenuto dell'Accumulatore.

CAMMINI DI ESECUZIONE DELL'ISTRUZIONE CONDIZIONATA

Nella seconda sequenza di istruzioni, se il valore immediato dell'operando è minore o uguale del contenuto dell'Accumulatore, l'istruzione JP P provoca un salto ad una istruzione seguente etichettata OK. I percorsi reali dell'esecuzione del programma della seconda sequenza di istruzioni possono sembrare un po' confusi se voi siete nuovi alla programmazione; **illustriamo** quindi **i percorsi dell'esecuzione come segue:**

```
        IN    A,(0)        ; Il carattere ASCII entra nell'Accumulatore
        RES   7,A          ; Azzera il bit di peso maggiore
; Confronta il codice ASCII col più basso codice legittimo
        CP    20H
        JP    P,OK         ; Se il codice è 20H o maggiore, fare il test per il
                           ; limite superiore
; Il codice è illegittimo. Supporre una densità 3
NOK:    LD    A,6          ; Carica la densità doppia
        JP    NEXT
; Confronta il codice ASCII col valore legittimo più grande
OK:     CP    7BH          ; Se il codice è 7BH o maggiore. Supporre una
        JP    P,NOK        ; densità 3
; Il codice ASCII è valido
        SUB   20H          ; Sottrai 20H
        LD    HL,INDX      ; Carica in HL l'indirizzo di base della tabella degli
                           ; indici
        ADD   L            ; Addiziona il contenuto dell'Accumulatore ad HL
        LD    L,A
        LD    A,(HL)       ; Carica l'indice nell'Accumulatore
        ADD   A            ; Moltiplica per 2
NEXT:   LD    HL,DELY      ; Carica l'indirizzo base della tabella dei ritardi
                           ; in HL
```


I percorsi dell'esecuzione, illustrati sopra da lettere cerchiate, si possono interpretare come segue:

Ⓐ Un codice ASCII supera il test "il valore legittimo più basso", ma ora deve essere verificato per "il valore legittimo più alto".

Ⓑ Il codice ASCII fallisce il test "il valore legittimo più basso". Il programma carica la densità doppia nell'Accumulatore e salta alla sequenza di istruzioni che accede alla costante di ritardo appropriata a questa densità. Questo salto è illustrato da Ⓒ .

Ⓓ Un carattere che ha superato il test "il valore ASCII legittimo più basso" viene successivamente verificato per "il valore legale ASCII più alto"; se questo test fallisce allora l'esecuzione del programma salta, come mostrato in Ⓔ , alle istruzioni che suppongono una densità pari a 3. Ⓔ , in realtà si accorda con Ⓑ

Ⓕ Un carattere ASCII che supera sia il test "il valore legittimo più basso" che il test "il valore legittimo più alto" è elaborato col percorso delle istruzioni Ⓕ . Le istruzioni in questo percorso caricano l'appropriato indice di densità nell'Accumulatore.

RESET E INIZIALIZZAZIONE

Per completare il nostro programma, dobbiamo creare le necessarie istruzioni di Reset e di Inizializzazione.

Si eseguono le istruzioni di Reset ogni volta che $\overline{\text{RESET}}$ è basso in ingresso del sistema del microcalcolatore.

Le istruzioni di inizializzazione saranno eseguite ogni volta che si dà il via al sistema.

Non c'è alcuna ragione perchè le sequenze di istruzioni di Reset e di Inizializzazione debbano coincidere; in molte applicazioni si può aver bisogno di due sequenze di istruzioni separate e distinte. **D'altra parte è molto comune usare il Reset in luogo di**

una inizializzazione del sistema. Ciò significa che quando si dà tensione al sistema la prima volta, si ha un impulso di $\overline{RESET}$; questo fa partire l'intero sistema di logica basato sul microcalcolatore.

Nel nostro caso il programma di Reset è veramente semplice. Tutto ciò che dobbiamo fare è di far uscire i codici di Controllo all'Interfaccia Parallela di Ingresso/Uscita dello Z80, quindi posizionare i segnali d'uscita per le condizioni "tra due cicli di stampa". **Ecco la necessaria sequenza di istruzioni d'inizializzazione:**

```
            ORG  0
; Facciamo dapprima uscire i codici di controllo verso il registro di controllo della
; Porta di I/O A
            LD    A,0FFH       ; Posiziona il modo 3
            OUT   (1),A
            OUT   (1),A        ; Tutte le linee sono ingressi
; Facciamo poi uscire i codici di controllo verso il registro di controllo della Porta
, di I/O B
            OUT   (3),A        ; Posiziona il modo 3
            LD    A,0F0H       ; Posiziona i pin da 0 a 3 in uscita ed i pin da
            OUT   (3),A        ; 4 a 7 in ingresso
; Posiziona alti HAMMER PULSE, PW READY e PW REL
; Posiziona basso START RIBBON MOTION
            LD    A,7
            OUT   (2),A
```

Ecco come si costruiscono i codici di controllo per ogni porta dello Z80 PIO:

Dopo aver posizionato il Modo 3, si deve scrivere un altro byte nel registro di Controllo della porta; questo secondo byte specifica la direzione di ogni pin della porta. Un bit ad 1 nel byte di direzione specifica una linea di ingresso e un bit a 0 specifica una linea di uscita.

SOMMARIO DEL PROGRAMMA

Prima di tutto sarebbe una buona idea mettere insieme l'intero programma, come sviluppato in questo capitolo. Includeremo le direttive Assembler necessarie. Questo programma finale è illustrato in Figura 4-6.

Ora che il programma è finito, noterete che la memoria RAM non è stata usata. I registri della CPU hanno fornito memoria di lettura e scrittura sufficienti per maneggiare tutti i dati variabili.

I 1024 byte della memoria programma in ROM sono sufficienti per contenere l'intero programma più le due tabelle dei dati.

Dopo l'implementazione di un sistema a microcalcolatore nei confini limitati della logica inclusa in questo capitolo, si potrebbero eliminare i due chip di memoria RAM. Con tutta probabilità, ci sarebbero numerose altre funzioni logiche che potrebbero essere incluse più economicamente nel sistema a microcalcolatore, queste richiederebbero quasi certamente la presenza di qualche memoria RAM. Ci sono nove byte di memoria a lettura e scrittura forniti dai sette registri della CPU e dal Puntatore della Catasta della CPU (Stack Pointer); questi sono usualmente insufficienti per una reale applicazione.

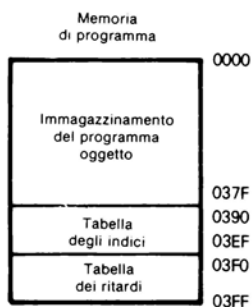
Ecco la mappa finale della memoria programmi, identificante il modo impiegato dal programma illustrato in Figura 4-6 per usare la memoria ROM:

```
INDX:     EQU  390H          ; Indirizzo base della tabella degli indici
DELY:     EQU  3F2H          ; Prima locazione nella tabella dei ritardi
          ORG  0
; Facciamo dapprima uscire i codici di controllo verso il registro di controllo della
; Porta di I/O A
          LD   A,0FFH        ; Posiziona il Modo 3
          OUT  (1),A
          OUT  (1),A         ; Tutte le linee sono ingressi
, Facciamo poi uscire i codici di controllo verso il registro di controllo della Porta di I/O B
          OUT  (3),A         ; Posiziona il Modo 3
          LD   A,0F0H        ; Posiziona i pin da 0 a 3 in uscita
          OUT  (3),A         ; ed i pin da 4 a 7 in ingresso
; Posiziona alti HAMMER PULSE, PW READY e PW REL
; Posiziona basso START RIBBON MOTION
          LD   A,7
          OUT  (2),A
; Programma di un ciclo di stampa
; Tra due cicli di stampa verifica su FFI (bit 5 della Porta di I/O B) per un valore zero
START:    IN   A,(2)         ; La Porta di I/O B entra nell'Accumulatore
          BIT  5,A           ; Test sul bit 5
          JR   NZ,START      ; Se non è zero, tornare a START
; Inizializzazione del ciclo di stampa. Posizionare 0 sui bit 0 ed 1 della Porta di I/O B,
; posizionare 1 sui bit 2 e 3 della Porta di I/O B
          LD   A,0CH         ; Carica la maschera nell'Accumulatore
          OUT  (2),A         ; Uscita verso la Porta di I/O B
```

Figura 4-6. Un semplice programma di un ciclo di stampa (segue)

```
; Posizionamento a 0 del bit 3 della Porta di I/O B, completando l'impulso START
; RIBBON MOTION
            RES   3,A           ; Azzeramento del bit 3 della maschera nell'Ac-
                                ; cumulatore
            OUT   (2),A         ; Uscita della Porta di I/O B
; Test sulla fine del posizionamento della ruota di stampa
; Il bit 5 della Porta di I/O B (FFI) sarà 1
PWPOS:      IN    A,(2)         ; La Porta di I/O B entra nell'Accumulatore
            BIT   5,A           ; Test sul bit 7 (HAMMER ENABLE)
            JR    Z,PWPOS       ; Se è 0, tornare ancora al test
, Esecuzione del ritardo di 2 millisecondi di stabilizzazione della ruota di stampa
            LD    A,0FAH        ; Carica la costante iniziale del ritardo di tempo
PWSET:      DEC   A             ; Decremento dell'Accumulatore
            JR    NZ,PWSET      ; Ridecremento dell'Accumulatore se non è zero
; Test sulle condizioni per l'alimentazione del martelletto
PHFIR:      IN    A,(2)         ; La Porta di I/O B entra nell'Accumulatore
            BIT   7,A           ; Test sul bit 7 (HAMMER PULSE)
            JP    Z,PWRDY       ; Se è 0, superare l'alimentazione del martelletto
            BIT   4,A           ; Test su HAMMER INTERLOCK
            JR    Z,PHFIR       ; Attesa per un valore diverso da zero prima della
                                ; alimentazione
; Alimentazione del martelletto
            RES   2,A           ; Posizionamento a basso di HAMMER PULSE
            OUT   (2),A         ; Uscita di uno 0 sul bit 2 della Porta di I/O B
            IN    A,(0)         ; Il carattere ASCII entra nell'Accumulatore
            RES   7,A           ; Azzeramento del bit di ordine maggiore
; Confronto del codice ASCII col valore legittimo più basso
            CP    20H
            JP    M,PWRDY       ; Se il codice è 1FH o minore, superare l'alimenta-
                                ; zione del martelletto
; Confronta il codice ASCII col valore legittimo più alto
            CP    7BH
            JP    P,PWRDY       ; Se il codice è 7BH o maggiore, superare l'alimen-
                                ; tazione del martelletto
; Il codice ASCII è valido
            SUB   20H           ; Sottrai 20H
            LD    HL,INDX       ; Carica l'indirizzo di base della tabella degli indici
                                ; in HL
            ADD   L             ; Somma il contenuto dell'Accumulatore ad HL
            LD    L,A
            LD    A,(HL)        ; Carica l'indice nell'Accumulatore
            ADD   A             ; Moltiplica per 2
            LD    HL,DELY       ; Carica l'indirizzo di base della tabella dei ritardi
                                ; in HL
            ADD   L             ; Somma il contenuto dell'Accumulatore ad HL
            LD    L,A
            LD    E,(HL)        ; Carica la costante del ritardo in D, E
            INC   HL
            LD    D,(HL)
PRDLY:      DEC   DE            ; Esegui il ritardo di stampa
            LD    A,D
```

Figura 4-6. Un semplice programma di un ciclo di stampa (segue)

```
           OR    E
           JR    NZ,PRDLY
           IN    A,(2)          ; Alla fine del ritardo far uscire 1 sul bit 2 della Porta
           SET   2,A            ; di I/O B. Ciò posiziona alto HAMMER PULSE
           OUT   (2),A
; Esecuzione di un ritardo di tempo di 3 millisecondi per PRINTWHEEL RELEASE
           LD    DE,231         ; Carica la costante iniziale del ritardo di tempo
PWREL:     DEC   DE             ; Esecuzione del ritardo di tempo lungo
           LD    A,D
           OR    E
           JR    NZ,PWREL
; Uscita di un 1 sul bit 0 della Porta di I/O B. Ciò posiziona alto PW REL
           IN    A,(2)          ; La Porta di I/O B entra nell'Accumulatore
           SET   0,A            ; Posizionamento ad 1 del bit 0
           OUT   (2),A          ; Uscita del risultato
; Esecuzione di un ritardo di tempo di 2 millisecondi per rendere pronta la ruota di
; stampa
PWRDY:     LD    A,0FAH         ; Carica la costante iniziale del ritardo di tempo
RDYDLY:    DEC   A              ; Decremento dell'Accumulatore
           JR    NZ,RDYDLY      ; Ridecremento se non è zero
; Test per EOR DET uguale a 1 (bit 6 della Porta di I/O B) come prerequisito per
; finire il ciclo di stampa
EORCHK:    IN    A,(2)          ; La Porta di I/O B entra nell'Accumulatore
           BIT   6,A            ; Test sul bit 6
           JR    Z,EORCHK       ; Se è 0 ritorna e ripeti il test
; Alla fine del ciclo di stampa posizionare ad 1 il bit 1 della Porta di I/O B ciò posi-
; ziona alto CH RDY
           SET   1,A            ; Posizionamento ad 1 del bit 1 della Porta B (nel-
                                ; l'Accumulatore)
           OUT   (2),A          ; Uscita del risultato
           JP    START          ; Salto al test del nuovo ciclo di stampa
; Qui segue la tabella degli indici
           ORG   INDX
           Qui seguono i dati rappresentanti gli ingressi di 90 indici
; Qui segue la tabella dei ritardi
           ORG   DELY
           Qui seguono i dati rappresentanti i 6 ritardi
```

Figura 4-6. Un semplice programma di un ciclo di stampa (conclusione)

Capitolo 5

PROSPETTIVA DEL PROGRAMMATORE

Il programma sviluppato nel Capitolo 4 è considerevolmente più breve e più facile da seguire di quello della simulazione digitale del Capitolo 3. Nonostante siamo arrivati al Capitolo 4 attraverso un lungo cammino, abbiamo ancora una via da percorrere. Il programma di Figura 4-6 tratta la logica da implementare come una singola funzione di trasferimento, ma esso non è un programma scritto bene.

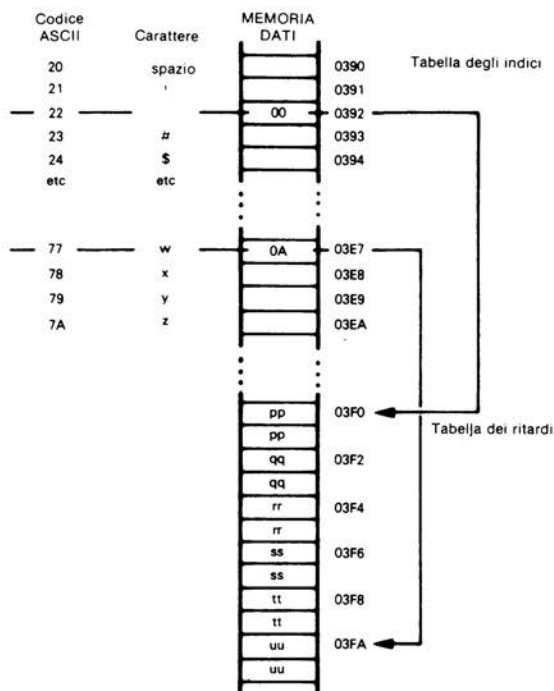
Per il progettista di logica digitale, una delle cose che maggiormente confonde a proposito della programmazione è la banale facilità con cui si può fare la stessa cosa in dieci modi diversi. Ciò implica che alcune implementazioni siano più efficienti di altre? In realtà sì. Scrivere programmi ad un alto grado di efficienza è una attitudine proprio come creare una efficiente logica digitale; ma ci sono alcune regole che, se seguite, vi aiuteranno almeno ad evitare errori evidenti. In questo capitolo prenderemo il programma creato nel Capitolo 4 e lo guarderemo un po' più attentamente.

EFFICIENZA DI UNA PROGRAMMAZIONE SEMPLICE

La prima cosa che dovrete fare, dopo aver scritto un programma sorgente, è di tornarci sopra, guardando i modi elementari con cui si possono eliminare delle istruzioni.

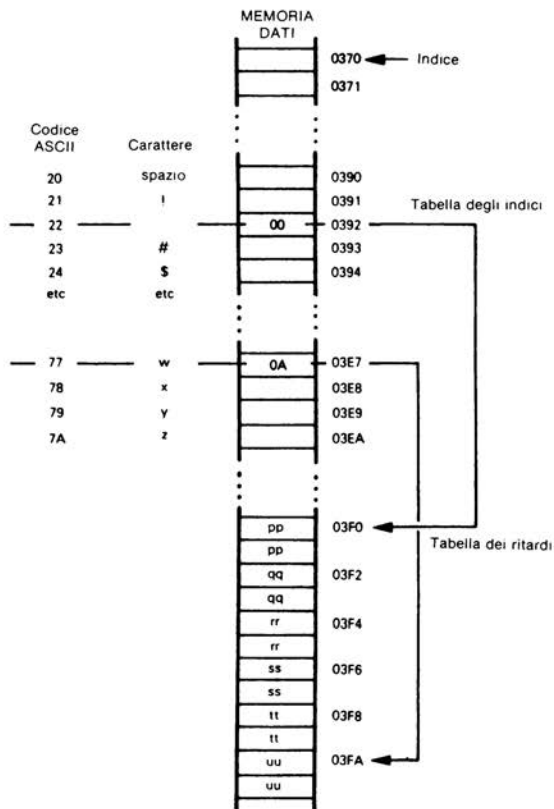
RICERCHE TABELLARI EFFICIENTI

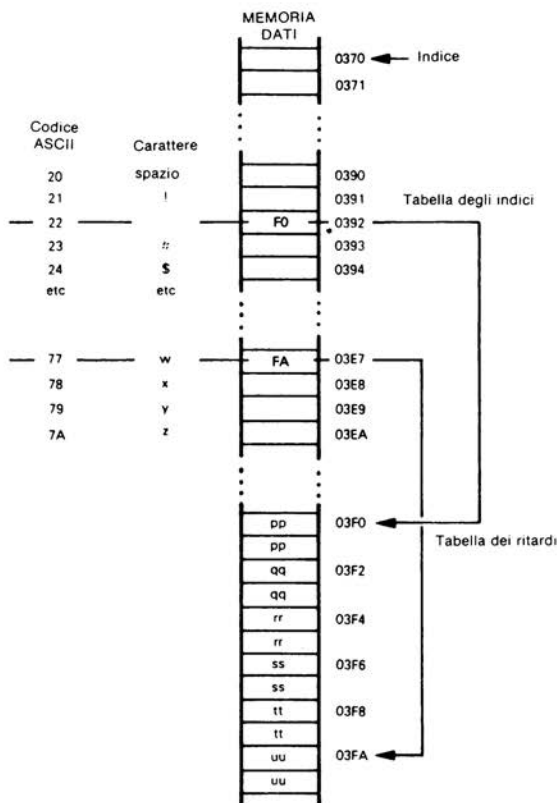
Mediamente, troverete che è possibile ridurre un programma a due terzi della sua lunghezza originale, scrivendo semplicemente sequenze di istruzioni più efficienti. In Figura 4-6 l'esempio più evidente di programmazione superficiale interessa la Tabella degli Indici. Il programma carica un valore tra 1 e 6 da un byte della Tabella indice, poi moltiplica questo valore per due, prima di sommarlo all'indirizzo base della Tabella dei Ritardi. **Perchè non memorizzare due volte l'indice nella Tabella degli Indici?**

Ciò elimina una istruzione come segue:

```
            —
            —
            —
; Il codice ASCII è valido
            SUB   20H          ; Sottrazione di 20H
            LD    HL,INDX      ; Carica l'indirizzo di base della tabella degli indi-
                               ; ci in HL
            ADD   L            ; Addiziona il contenuto dell'Accumulatore ad HL
            LD    L,A
            LD    A,(HL)       ; Carica nell'Accumulatore l'Indice X2
            LD    HL,DELY      ; Carica l'indirizzo di base della tabella dei ritar-
                               ; di in HL
            ADD   L            ; Addiziona il contenuto dell'Accumulatore ad HL
            LD    L,A
            —
            —
            —
l'istruzione
ADD è abolita
```

Nella precedente sequenza d'istruzioni, è da notare che si è abolita una sola istruzione dopo l'istruzione LD ombreggiata.

Ci sono ancora numerosi altri modi coi quali fare una supervisione più efficiente della Tabella dei Ritardi. Per esempio, **perchè si sottrae 20H dal codice ASCII?** Se vogliamo

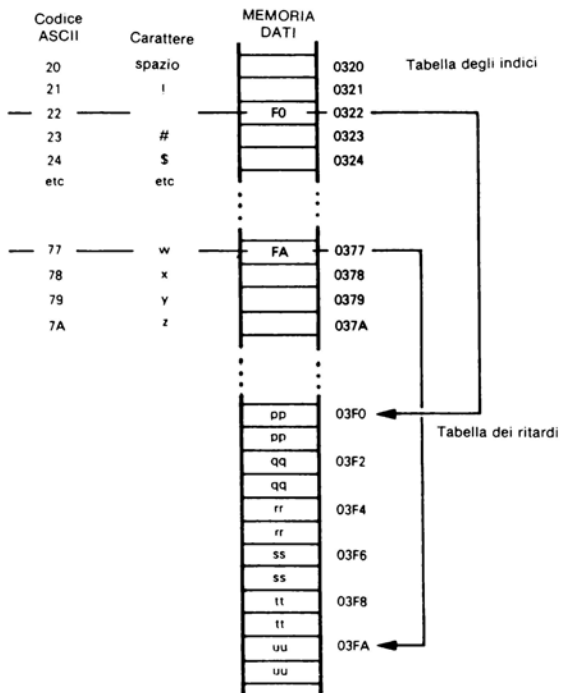
sommare il codice ASCII ad un indirizzo di base, non c'è niente che ci impedisca di fare una istruzione Equating (EQU) di questo indirizzo di base, rappresentato dal simbolo INDEX, con il valore 20_{16} meno il primo byte reale della Tabella degli Indici. La nostra sequenza di istruzioni crolla ulteriormente come segue:

```
INDEX:    EQU  0370H        ; Uguaglia l'indirizzo base della Tabella degli Indici
                            ; – 20H
          –
          –
          –
; Il codice ASCII è valido
          LD   HL,INDX      ; Carica l'indirizzo base della Tabella degli Indici
                            ; – 20H
          ADD  L            ; Addiziona il contenuto dell'Accumulatore ad HL
          LD   L,A
          LD   A,(HL)       ; Carica nell'Accumulatore l'Indice X2
          LD   HL,DELY      ; Carica l'indirizzo di base della Tabella dei Ritardi
                            ; in HL
l'istruzione ADD  L         ; Addiziona il contenuto dell'Accumulatore ad HL
SUB è     LD   L,A
abolita
```

Ora INDEX è uguale a 0370_{16} – che significa che non dobbiamo più sottrarre 20_{16} dal codice ASCII. Abbiamo eliminato l'istruzione SUB che prima era l'istruzione LD ombreggiata. **Ora invece di memorizzare due volte l'indice della densità del carattere nella Tabella degli Indici, perchè non memorizzare la seconda metà dell'indirizzo della Tabella dei Ritardi?** Il nostro programma ora si contrae ulteriormente come segue:

```
INDEX:  EQU  0370H        ; Uguaglia l'indirizzo base della Tabella degli Indici
                          ; – 20H
        –
        –
        –
; Il codice ASCII è valido
        LD   HL,INDX      ; Carica l'indirizzo base della Tabella degli Indici
                          ; – 20H
        ADD  L            ; Addiziona il contenuto dell'Accumulatore ad HL
        LD   L,A
        LD   L,(HL)       ; Carica il byte di ordine minore dell'indirizzo della
                          ; Tabella dei Ritardi
        LD   H,3          ; Carica il byte di ordine maggiore dell'indirizzo
                          ; della Tabella dei Ritardi
```

Sono scomparse altre due istruzioni.

Abbiamo ora tolto quattro istruzioni dalla sequenza che carica la costante di ritardo iniziale dell'alimentazione del martelletto — e non abbiamo ancora finito.

TABELLE COLLOCATE IN POSIZIONE GIUSTA PER SEMPLIFICARE LA SEQUENZA DI ACCESSO ALLE ISTRUZIONI

Perchè non spostiamo l'intera Tabella degli Indici, cosicchè invece di occupare le locazioni di memoria da 0390_{16} a $03EA_{16}$, essa occuperebbe le locazioni di memoria da 0320_{16} a $037A_{16}$? Il codice ASCII, privato del bit di parità, diventa ora il byte di ordine minore dell'indirizzo della Tabella degli Indici: La nostra sequenza di istruzioni si contrae ulteriormente come segue:

```
; Il codice ASCII è valido
        LD    H,3          ; Carica il byte di ordine maggiore dell'indirizzo
                           ; della Tabella degli Indici
        LD    L,A          ; Sposta il byte di ordine minore dell'indirizzo
                           ; in L
        LD    L,(HL)       ; Carica il byte di ordine minore dell'indirizzo della
                           ; Tabella dei Ritardi
```

Supponiamo che si debba stampare un carattere "w". Prima di eseguire la prima delle tre precedenti istruzioni, l'Accumulatore contiene 77_{16}, come risultato dell'esecuzio-

ne delle istruzioni precedenti:

```
IN   A,(0)
RES  7,A
```

Dopo l'esecuzione dell'istruzione:

```
LD   H,3
```

il registro H conterrà 03_{16}; questo è la metà superiore dell'indirizzo implicito di memoria. Successivamente l'istruzione:

```
LD   L,A
```

sposta 77_{16} dall'Accumulatore al registro L. H ed L contengono ora 0377_{16}, che è l'effettivo indirizzo implicato. L'istruzione successiva:

```
LD   L,(HL)
```

sposta il contenuto del byte di memoria indirizzata da HL nel registro L

HL contiene 0377_{16}. Il byte di memoria 0377_{16} contiene FA_{16}, quindi si sposta FA_{16} nel registro L. Il nuovo indirizzo implicito è $03FA_{16}$; e questo è l'indirizzo richiesto della Tabella dei Ritardi.

Nove istruzioni sono state ridotte a tre e il solo prezzo pagato è stato quello di spostare la Tabella degli Indici in una nuova area di memoria dati.

Per assicurarci che abbiamo capito come apparirà ora il programma, si mostrano sotto la nuova e la vecchia sequenza di istruzioni, una di fronte all'altra, senza i campi dei commenti:

```
      Programma vecchic          Programma nuovo
IL CODICE ASCII È VALIDO
         SUB     20H             LD      H,3
         LD      HL,INDX         LD      L,A
         ADD     L               LD      L,(HL)
         LD      L,A
         LD      A,(HL)
         ADD     A
         LD      HL,DELY
         ADD     L
         LD      L,A
```

Sfortunatamente non esistono regole d'oro che, se seguite, assicureranno che scriverete sempre il programma più corto possibile. Una volta che avete scritto alcuni programmi, capirete come funzionano le istruzioni individualmente e che ciò a sua volta, genera efficienza. Lo scopo delle pagine precedenti è stato quello di dimostrare l'enorme differenza tra un programma compatto e un programma diretto. Se il vostro prodotto deve essere prodotto in grande volume, è necessario che spendiate tempo e danaro per ridurre la dimensione dei programmi — sarete allora capaci di eliminare alcuni dei vostri chip di ROM..

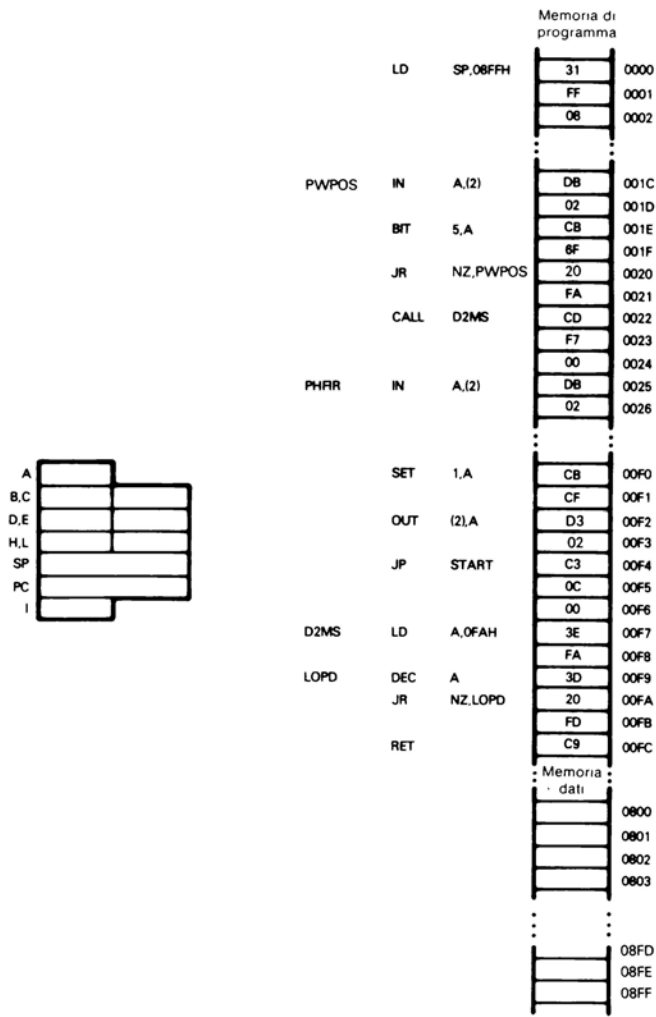
SOTTOPROGRAMMI

Se riguardate il programma di Figura 4–6, noterete che in due punti di questo programma si eseguono sequenze d'istruzioni identiche per creare un ritardo di due millisecondi. Ora, occorrono solo tre istruzioni per eseguire un ritardo di due millisecondi, cosicchè, il fatto di ripetere queste tre istruzioni, non costituisce una grande tragedia. Tuttavia, pensandoci, esiste la possibilità per una utilizzazione di memoria molto poco economica in programmi più lunghi.

Abbiamo mantenuto semplice il nostro programma nel Capitolo 4, perchè rimanesse abbastanza piccolo da poter essere maneggiato in un libro; ma, se volete, progettate una routine più complessa dove si deve ripetere una sequenza di 30 istruzioni, piuttosto che una sequenza di tre istruzioni. Dobbiamo ora trovare un modo per includere la sequenza di istruzioni una volta sola, quindi saltare a questa singola sequenza da un numero di locazioni diverse in un programma, a seconda del bisogno. Ciò sarà fatto da un sottoprogramma.

Prendiamo le tre istruzioni che eseguono un ritardo di due millisecondi e convertiamolo in un sottoprogramma. Ciò avviene per parti rilevanti del programma:

```
        ORG  0
        LD   SP,08FFH       ; Inizializzazione del puntatore della catasta alla fi-
                            ; ne dell'area dei dati
        —
        —
        —
; Esecuzione del ritardo di due millisecondi di posizionamento della ruota di stampa
        CALL D2MS
        —
        —
        —
; Esecuzione di un ritardo di due millisecondi di ruota di stampa pronta
PWRDY   CALL D2MS
        —
        —
        —
; Alla fine del ciclo di stampa posizionare ad 1 il bit 1 della Porta di I/O B ciò posi-
; ziona alto CH RDY
        SET  1,A
        OUT  (2),A
        JP   START
; Sottoprogramma per eseguire un ritardo di 2 millisecondi
D2MS    LD   A,0FAH         ; Carica 0 nell'Accumulatore
LOPD    DEC  A              ; Decrementa A
        JR   NZ,LOPD        ; Se A non è decrementato a 0, ridecrementalo
        RET                 ; Ritorno dal sottoprogramma
```

Per capire come funziona un sottoprogramma, assegneremo alcuni indirizzi di memoria arbitrari al codice oggetto del nostro programma sorgente; mostreremo, passo a passo, che cosa succede quando si chiama un sottoprogramma e che cosa succede quando si ritorna dal sottoprogramma. **Anzitutto ecco la mappa di memoria supposta:**

CHIAMATA DEL SOTTOPROGRAMMA

Supponiamo che si stia per eseguire la prima istruzione CALL D2MS. A questo punto i registri conterranno i dati seguenti:

Registro		
A	00	
B C		
D E		
H L		
SP	08FF	
PC	0022	
I	28	

Memoria di programma

	LD	SP 08FFH	31	0000
			FF	0001
			08	0002
			⋮	
PWPOS	IN	A,(2)	DB	001C
			02	001D
	BIT	5,A	CB	001E
			6F	001F
	JR	NZ,PWPOS	20	0020
			FA	0021
	CALL	D2MS	CD	0022
			F7	0023
			00	0024
PHFIR	IN	A,(2)	DB	0025
			02	0026
			⋮	
	SET	1,A	CB	00F0
			CF	00F1
	OUT	(2),A	D3	00F2
			02	00F3
	JP	START	C3	00F4
			0C	00F5
			00	00F6
D2MS	LD	A 0FAH	3E	00F7
			FA	00F8
LOPD	DEC	A	3D	00F9
	JR	NZ,LOPD	20	00FA
			FD	00FB
	RET		C9	00FC
			DATA MEMORY	
				0800
				0801
				0802
				0803
			⋮	
				08FD
				08FE
				08FF

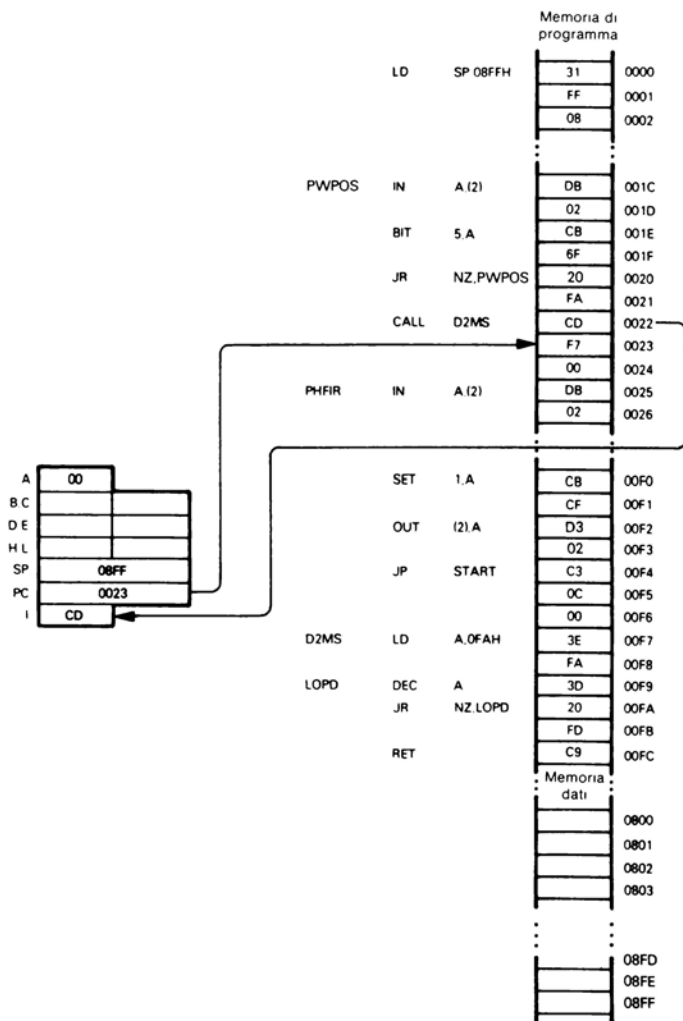
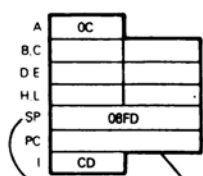
Il Contatore di Programma (PC) indirizza il primo byte del codice oggetto dell'istruzione Call; questo indirizzo è 0022_{16}. Il registro delle Istruzioni contiene il codice oggetto dell'istruzione eseguita più recentemente; questa è una istruzione JR allocata nel byte 0020_{16}. Il Puntatore della Catasta era stato inizializzato all'inizio del programma; esso contiene $08FF_{16}$. In accordo con la Figura 4-2 questo è l'indirizzo

del primo byte della memoria di lettura e scrittura. Poichè la Catasta (Stack) non è stata usata, il Puntatore della Catasta conterrà ancora $08FF_{16}$.

L'Accumulatore contiene 00 perchè questa era la condizione che faceva sì che l'esecuzione uscisse dal loop di mantenimento e partisse da PWPOS.

Ora dopo l'esecuzione dell'istruzione Call, si susseguono i seguenti passi:

Il codice oggetto dell'istruzione Call è caricato nel registro delle Istruzioni e si incrementa il Contatore di Programma:

Il Contatcre di Programma è incrementato di due per superare l'indirizzo della CALL. Questo valore incrementato è salvato nei primi due byte della catasta. L'indirizzo della CALL è quindi caricato nel Contatore dei Programmi. Il Puntatore della Catasta è decrementato di due per indirizzare il primo byte libero della catasta:

La successiva istruzione eseguita ha il suo codice oggetto memorizzato nel byte di memoria $00F7_{16}$; questo è il byte di memoria indirizzato ora dal Contatore di Programma:

Si eseguono ora le istruzioni del loop del ritardo di due millisecondi ripetutamente finchè il contenuto dell'Accumulatore si decrementi da 01 a 00.

RITORNO DAL SOTTOPROGRAMMA

Quando finalmente l'Accumulatore si decrementa da 01 a 00, l'esecuziona passa all'istruzione Return (RET). Questa istruzione incrementa il contenuto del Puntatore della Catasta di 2, quindi sposta il contenuto dei due byte superiori della Catasta nel Contatore di Programma. In tale modo l'esecuzione del programma ritorna all'istruzione che segue la Call:

In sommario ecco che cosa è successo:

Quando si è eseguita l'istruzione Call, si è salvato l'indirizzo dell'istruzione successiva nella catasta. L'istruzione Call ha fornito l'indirizzo dell'istruzione che deve essere eseguita successivamente.

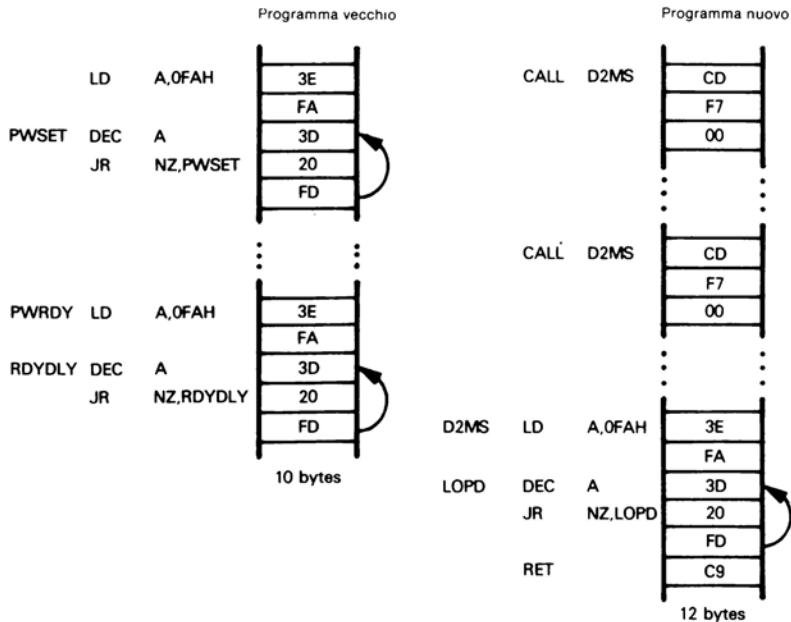
Successivamente si è eseguita la prima istruzione del sottoprogramma.

L'ultima istruzione del sottoprogramma ha solamente fatto sì che l'indirizzo salvato sulla sommità della catasta ritornasse nel Contatore di Programma e questo, a sua volta, ha provocato il salto all'esecuzione dell'istruzione seguente la Call.

QUANDO SI USANO I SOTTOPROGRAMMI

C'è un prezzo associato all'uso di sottoprogrammi:

1) **Ogni istruzione Call rappresenta tre byte aggiuntivi di codice oggetto.**
2) **La sequenza di istruzioni che deve essere spostata al sottoprogramma deve avere alla fine un'istruzione Return che costa un byte di codice oggetto.**

Guardiamo ora il nostro caso specifico. Le tre istruzioni che costituiscono il ritardo di due millisecondi occupano cinque byte di codice oggetto. Queste tre istruzioni si ripetono due volte, perciò, combinate, esse occupano dieci byte di codice oggetto. Quando si sposta in un sottoprogramma, l'aggiunta dell'istruzione Return alimenta i byte di codice oggetto da cinque a sei. Inoltre ci sono due istruzioni Call ognuna delle quali richiede tre byte di codice oggetto — ciò significa che le due istruzioni Call più il sottoprogramma generano dodici byte di codice oggetto. Ciò può essere illustrato come segue:

Nel nostro caso specifico, perciò, lo spostamento della sequenza di istruzioni del ritardo di due millisecondi in un sottoprogramma ci è costato due byte di codice

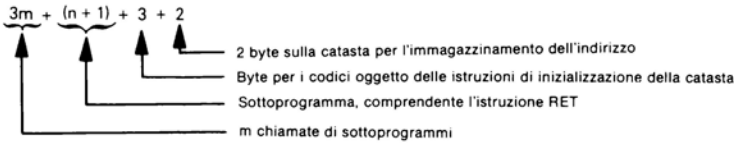
oggetto. Ci è costato tre byte di codice oggetto addizionali — quelli richiesti per inizializzare il Puntatore della Catasta; il nostro sistema a microcalcolatore richiederà ora memoria RAM.

Una catasta può esistere solo se è presente della memoria a lettura e scrittura.

Questi commenti non implicano che i sottoprogrammi abbiano una caratteristica di programmazione incerta, tale da essere usati avaramente; al contrario è arduo concepire un programma che, se ben scritto, non contenga qualche sottoprogramma. Ma tenete in mente che **esiste una dimensione minima del sottoprogramma al di sotto della quale i sottoprogrammi, in generale, diventano antieconomici.**

Supponiamo che ci siano n byte di codice oggetto in una sequenza di istruzioni che state pensando di convertire in un sottoprogramma.

Supponiamo che **gli n byte di codice oggetto si ripetano m volte;** ciò significa che quando gli n byte di codice oggetto diventano un sottoprogramma, essi saranno chiamati da m istruzioni CALL.

Senza sottoprogrammi, si consumeranno m x n byte ripetendo n byte m volte.
Con sottoprogrammi il numero di byte consumato è:

Per il sottoprogramma che vale la pena di essere fatto, 3m + n + 6 deve essere minore di m x n.

La Tabella 5-1 mostra la minima lunghezza economica di sottoprogramma in funzione del numero di chiamate di sottoprogramma.

Tabella 5-1. La lunghezza più breve di un sottoprogramma economico, come funzione del numero di volte con cui il sottoprogramma è chiamato

Numero di chiamate di sottoprogrammi (m)	Lunghezza Minima Economica del Sottoprogramma (n)
2	12 Bytes
3	8 Bytes
4	6 Bytes
5	6 Bytes
10	4 Bytes
20	4 Bytes

RITORNI CONDIZIONATI DA SOTTOPROGRAMMI

Anche se nessuna delle sequenze d'istruzioni ripetute nel programma di Figura 4-6 è abbastanza lunga da giustificare il suo mutamento in sottoprogramma, nondimeno esploriamo ulteriormente la potenza dei sottoprogrammi.

Così come esistono istruzioni di Jump condizionato, che si usano frequentemente in un loop di ritardi di tempo, esistono pure istruzioni di Call di un sottoprogramma condizionate e istruzioni di Return da Sottoprogrammi condizionate.

Istruzioni condizionate di Call e di Return, di e da Sottoprogrammi, sono particolarmente utili in sottoprogrammi più lunghi nei quali ci siano percorsi di esecuzione variabili.

Consideriamo la sequenza d'istruzioni che alimenta il martelletto di stampa di Figura 4-6. Dato il programma illustrato, questa sequenza di istruzioni si verifica una volta sola, cosa che significa che la sua conversione in un sottoprogramma non avrebbe senso. **E' possibile immaginare un programma più esteso che realizza una grande varietà di operazioni di interfacciamento con la stampante, tali che la logica che alimenta il martelletto di stampa deve essere avviata per un diverso numero di ragioni.**

Poichè la logica che alimenta il martelletto di stampa consiste in un insieme di istruzioni abbastanza lungo, il mettere queste istruzioni in un sottoprogramma dovrebbe essere assolutamente obbligatorio. Consideriamo la seguente implementazione di sottoprogramma:

```
; Sottoprogramma che alimenta il martelletto di stampa
PHFIR:  IN    A,(2)     ; La Porta di I/O B entra nell'Accumulatore
        BIT   7,A       ; Test sul bit 7 (HAMMER ENABLE)
        RET   Z         ; Se è 0, ritorna
        BIT   4,A       ; Test su HAMMER INTERLOCK
        RET   Z         ; Se è 0, ritorna
; Alimentazione del martelletto di stampa
        RES   2,A       ; Posizionamento a basso di HAMMER PULSE
        OUT   (2),A     ; Uscita di uno 0 sul bit 2 della Porta di I/O B
        IN    A,(0)     ; Il carattere ASCII entra nell'Accumulatore
        RES   7,A       ; Azzera il bit di ordine maggiore
; Confronto del codice ASCII col valore legittimo minore
        CP    20H
        RET   M         ; Se il codice è 1FH o minore, superare l'alimenta-
                        ; zione del martelletto
; Confronto del codice ASCII col valore legittimo maggiore
        CP    7BH
        RET   P         ; Se il codice è 7BH o maggiore superare l'alimen-
                        ; tazione del martelletto
; Il codice ASCII è valido
        LD    H,03H     ; Carica il byte di ordine maggiore dell'indirizzo
                        ; della Tabella degli Indici
        LD    L,A       ; Sposta il byte di ordine minore dell'indirizzo in L
        LD    L,(HL)    ; Carica il byte di ordine minore dell'indirizzo della
                        ; Tabella dei Ritardi
        CALL  LDLY
        IN    A,(2)     ; Alla fine del ritardo far uscire un 1 sul bit 2 della
        SET   2,A       ; Porta di I/O B. Questo posiziona alto HAMMER
                        ; PULSE
        OUT   (2),A
, Esecuzione di un ritardo di tempo di tre millisecondi per PRINTWHEEL RELEASE
        LD    HL,MS3
        CALL  LDLY
; Uscita di un 1 sul bit 0 della Porta di I/O B. Questo posiziona alto PW REL
        IN    A,(2)     ; La Porta di I/O B entra nell'Accumulatore
        SET   0,A       ; Posizionamento ad 1 del bit 0
        OUT   (2),A     ; Uscita del risultato
        RET             ; Ritorno dal Sottoprogramma
; Sottoprogramma di ritardo lungo. Supponiamo che H ed L indirizzino il primo di
; due byte di un dato contenente la costante del ritardo iniziale
```

```
LDLY    LD    E,(HL)    ; Carica in D, E la costante del ritardo
        INC   HL
        LD    D,(HL)
LDLP:   DEC   DE        ; Esecuzione del ritardo di stampa
        LD    A,D
        OR    E
        JR    NZ,LDLP
        RET             ; Ritorno alla fine del ritardo lungo
MS3     DEFW  231       ; Costante del ritardo di tempo per PRINTWHEEL
                        ; RELEASE
```

RITORNO CONDIZIONATO

Il sottoprogramma illustrato sopra alimenta il martelletto di stampa se si realizzano tutte le condizioni necessarie; si esegue una uscita veloce se una delle condizioni non è verificata. Le istruzioni del Return condizionato sono ombreggiate.

SOTTOPROGRAMMI ANNIDATI

Notate che **abbiamo aggiunto un sottoprogramma nel sottoprogramma. La sequenza di istruzioni del ritardo lungo è stata spostata in un sottoprogramma, la prima istruzione del quale è etichettato LDLY. Ci si riferisce ad esso come a un "sottoprogramma annidato".**

PARAMETRI DI UN SOTTOPROGRAMMA

Un aspetto strano del sottoprogramma LDLY è che esso richiede la memorizzazione della costante di ritardo iniziale in due byte di memoria, il primo dei quali è indirizzato dai registri H ed L quando si chiama LDLY. **Le istruzioni nel sottoprogramma LDLY caricheranno realmente nei registri D ed E la costante del ritardo iniziale. La costante del ritardo iniziale diventa un parametro,** che permette ad un solo sottoprogramma di implementare uno spettro completo di ritardi di tempo. I parametri di un sottoprogramma sono una caratteristica molto importante nell'uso di sottoprogrammi.

La seconda volta che si chiama il sottoprogramma LDLY, invece di caricare la costante iniziale richiesta (231) nei registri D ed E, si carica un indirizzo rappresentato dal simbolo MS3 nei registri H ed L. Il simbolo MS3 diventerà l'indirizzo di un dato a due byte situato da qualche parte nella memoria; in questi due byte di deve memorizzare il valore 231.

RITORNI MULTIPLI DA UN SOTTOPROGRAMMA

Il sottoprogramma PHFIR non è così utile come potrebbe essere. Ci sono quattro ritorni condizionati da questo sottoprogramma, ognuno dei quali ha l'avvio da una differente condizione non verificata. C'è pure un ritorno da sottoprogramma in seguito ad un'alimentazione valida del martelletto di stampa.

Come fa il programma chiamante a conoscere se, dopo aver chiamato PHFIR, il martelletto di stampa è o non è stato alimentato? Non è molto sicuro fare dei test sugli stati, poichè non possiamo essere certi di che cosa capita alle condizioni degli stati durante l'esecuzione stessa delle istruzioni che alimentano il martelletto di stampa.

Sottoprogrammi contenenti un grande numero di uscite condizionate da errori, in aggiunta a un ritorno standard, conterranno spesso una logica che fa ritornare ad un numero differente di istruzioni del programma chiamante. Prendiamo il caso del sottoprogramma PHFIR. La sequenza di istruzioni che chiama questo sottoprogram-

ma può sembrare la seguente:

```
              —
              —
              —
RT0     CALL  PHFIR     ; Chiamata del sottoprogramma che alimenta il
                        ; martelletto di stampa
        JR    RT1       ; Ritornare qui per il riposizionamento della ruota
                        ; di stampa
        JR    RT0       ; Ritornare qui per un basso su HAMMER
                        ; INTERLOCK
        JR    RT2       ; Ritornare qui per un codice ASCII minore di 20H
        JR    RT3       ; Ritornare qui per un codice ASCII maggiore di
                        ; 7AH
; Le istruzioni che seguono sono eseguite dopo un'alimentazione valida del martel-
; letto di stampa
              —
              —
              —
; Le istruzioni che seguono sono eseguite per il riposizionamento della ruota di stampa
RT1     —
        —
        —
; Le istruzioni che seguono sono eseguite per un codice ASCII minore di 20H
RT2     —
        —
        —
; Le istruzioni che seguono sono eseguite per codice ASCII maggiore di 7AH
RT3     —
        —
        —
```

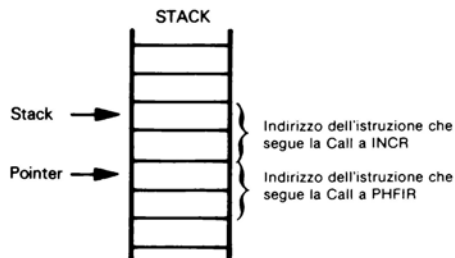
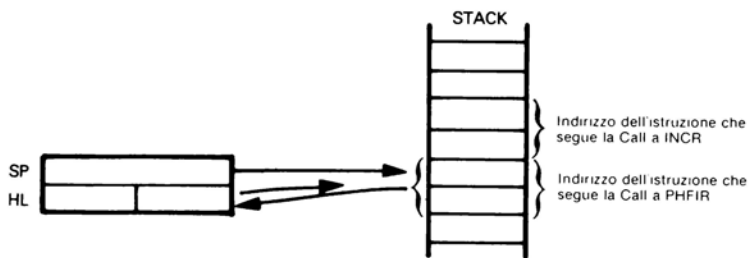
Per funzionare secondo questo schema, il sottoprogramma PHFIR deve incrementare l'indirizzo di ritorno, che è memorizzato nei due byte in cima alla catasta, ogni volta che si esegue un Return condizionato. Il sottoprogramma PHFIR è pertanto modificato come segue:

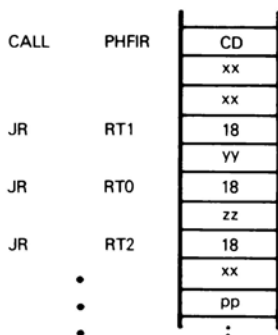
```
; Sottoprogramma che alimenta il martelletto di stampa
PHFIR:  IN    A,(2)     ; La Porta di I/O B entra nell'Accumulatore
        BIT   7,A       ; Test sul bit 7 (HAMMER ENABLE)
        RET   Z         ; Se è 0, ritorna
        CALL  INCR      ; Incrementa l'indirizzo di ritorno
        BIT   4,A       ; Test su HAMMER INTERLOCK
        RET   Z         ; Se è 0, ritorna
        CALL  INCR      ; Incrementa l'indirizzo di ritorno
; Alimentazione del martelletto di stampa
        RES   2,A       ; Posizionamento a basso di HAMMER PULSE
        OUT   (2),A     ; Uscita di uno 0 sul bit 2 della Porta di I/O B
        IN    A,(0)     ; Il carattere ASCII entra nell'Accumulatore
        RES   7,A       ; Azzera il bit di ordine maggiore
; Confronto del codice ASCII col valore legittimo minore
        CP    20H
        RET   M         ; Se il codice è 1FH o minore, superare l'alimenta-
                        ; zione del martelletto
        CALL  INCR      ; Incrementa l'indirizzo di ritorno
```

```
; Confronto del codice ASCII col valore legittimo maggiore
          CP      7BH
          RET     P          ; Se il codice è 7BH o maggiore, superare l'alimen-
                             ; tazione del martelletto
          CALL    INCR       ; Incrementa l'indirizzo di ritorno
; Il codice ASCII è valido
          LD      H,03H      ; Carica il byte di ordine maggiore dell'indirizzo
                             ; della Tabella degli Indici
          LD      L,A        ; Sposta il byte di ordine minore dell'indirizzo
          LD      L,(HL)     ; Carica il byte di ordine minore dell'indirizzo della
                             ; Tabella dei Ritardi
          CALL    LDLY
          IN      A,(2)      ; Alla fine del ritardo fa uscire un 1 sul bit 2 della
          SET     2,A        ; Porta di I/O B. Questo posiziona alto HAMMER
                             ; PULSE
          OUT     (2),A
; Esecuzione di un ritardo di tempo di tre milliscondi per PRINTWHEEL RELEASE
          LD      HL,MS3
          CALL    LDLY
; Uscita di un 1 sul bit 0 della Porta di I/O B. Questo posiziona alto PW REL
          IN      A,(2)      ; La Porta di I/O B entra nell'Accumulatore
          SET     0,A        ; Posizionamento ad 1 del bit 0
          OUT     (2),A      ; Uscita del risultato
          RET                ; Ritorno dal sottoprogramma
; Sottoprogramma di ritardo lungo. Supponiamo che H ed L indirizzino il primo di
; due byte di un dato contenente la costante del ritardo iniziale
LDLY      LD      E,(HL)     ; Carica in D, E la costante del ritardo
          INC     HL
          LD      D,(HL)
LDLP:     DEC     DE         ; Esecuzione del ritardo di stampa
          LD      A,D
          OR      E
          JR      NZ,LDLP
          RET                ; Ritorno alla fine del ritardo lungo
MS3       DEFW    231        ; Costante del ritardo di tempo per PRINTWHEEL
                             ; RELEASE
; Sottoprogramma per incrementare l'indirizzo di ritorno del sottoprogramma chia-
; mante
INCR      INC     SP         ; Incrementa due volte il puntatore della catasta
          INC     SP         ; per accedere all'indirizzo di ritorno di PHFIR
          EX      (SP),HL    ; Scambia HL con l'indirizzo di ritorno di PHFIR
          INC     HL         ; Somma di 2 all'indirizzo di ritorno
          INC     HL
          EX      (SP),HL    ; Rimemorizza l'indirizzo di ritorno
          DEC     SP         ; Decrementa due volte il puntatore di catasta
          DEC     SP
          RET                ; Ritorno
```

MANIPOLAZIONE DELLA CATASTA (STACK)

Il sottoprogramma INCR è interessante; esso mostra come si possa manipolare la catasta. Diamo un'occhiata a ciò che accade. Non appena si entra nel sottoprogramma INCR, si incrementa di due il contenuto del Puntatore della Catasta. Ciò ha l'effetto di indirizzare l'indirizzo di ritorno di PHFIR piuttosto

che l'indirizzo di ritorno di INCR:

L'istruzione EX (SP),HL salva semplicemente il contenuto dei registri H ed L alla sommità della catasta, mentre sposta ciò che era alla sommità della catasta nei registri H ed L:

Le due istruzioni successive sommano 2 al contenuto dei registri H ed L, che ora contengono l'indirizzo di ritorno di PHFIR. Sommiamo 2 all'indirizzo di ritorno, perchè, se guardate la sequenza di chiamata, essa è seguita da una serie di istruzioni di Jump (JR). Ogni istruzione JR occupa due byte , il che significa che, ogni volta che si supera un Conditional Return, dobbiamo incrementare l'indirizzo di ritorno di 2:

CALL	PHFIR	CD
		xx
		xx
JR	RT1	18
		yy
JR	RT0	18
		zz
JR	RT2	18
	•	xx
	•	pp
	•	:

Il successivo EX (SP),HL rimemorizza semplicemente l'indirizzo di ritorno incrementato, PHFIR in cima alla catasta.

In fine dobbiamo rimemorizzare nel Puntatore della Catasta il suo contenuto originale, cosicchè l'istruzione di Return INCR riporterà l'indirizzo di ritorno corretto.

CHIAMATE CONDIZIONATE A SOTTOPROGRAMMI

Creeremo un altro sottoprogramma che alimenta il martelletto di stampa, ma che non fa nessun test per assicurarsi che il martelletto di stampa sia stato alimentato. Questo sottoprogramma suppone semplicemente che nell'Accumulatore ci sia un carattere ASCII valido e che si debba alimentare il martelletto di stampa. Tutta la logica per determinare la validità dell'alimentazione del martelletto di stampa è esterna al sottoprogramma che fa alimentare il martelletto di stampa; perciò, questo sottoprogramma è chiamato in modo condizionato — dal momento che si sono verificate tutte le condizioni di alimentazione del martelletto di stampa. Ecco come appare ora il nostro programma:

```
; Test sulle condizioni di alimentazione del martelletto
PHFIR:  IN    A,(2)      ; La Porta di I/O B entra nell'Accumulatore
        BIT   7,A        ; Test sul bit 7 (HAMMER ENABLE)
        JP    Z,PWRDY    ; Se è 0, superare l'alimentazione del martelletto
        BIT   4,A        ; Test su HAMMER INTERLOCK
        JR    Z,PHFIR    ; Attesa di un valore non zero prima dell'alimen-
                         ; tazione
  Ingresso del carattere da stampare
        IN    A,(0)      ; Il carattere ASCII entra nell'Accumulatore
        RES   7,A        ; Azzeramento del bit di ordine maggiore
  Confronto del codice ASCII col valore legittimo minore
        CP    20H
        JP    M,PWRDY    ; Se il codice è 1FH o minore superare l'alimenta-
                         ; zione del martelletto
; Confronto del codice ASCII col valore legittimo più alto
        CP    7BH
        CALL  M,FIRE     ; Se il codice è valido, chiamare il sottoprogramma
                         ; di alimentazione
; Esecuzione del ritardo di 2 millisecondi PRINTWHEEL READY
PWRDY   LD    A,0FAH     ; Carica la costante del ritardo di tempo
        —
        —
        —
```

E' da notare come l'istruzione Conditional Return rifletta la logica di programmazione OR, mentre l'istruzione Conditional Call rifletta la logica AND. In tale modo il sottoprogramma PHFIR include un numero di istruzioni Conditional Return, ognuna delle quali sarà eseguita solo se si incontrano condizioni non valide. Il sottoprogramma FIRE, d'altro canto, è chiamato condizionatamente solo quando l'ultima delle condizioni valide necessarie sia stata verificata.

Il sottoprogramma FIRE non è mostrato in dettaglio, poichè la sua scrittura avrebbe aggiunto poco alla comprensione dell'istruzione Conditional Call. Con riferimento alla Figura 4-6, il sottoprogramma FIRE sarebbe stato formato da istruzioni del tipo.

Posizionare il segnale Hammer Pulse basso
Eseguire il ritardo dell'impulso che alimenta il martelletto
Posizionare alto l'impulso che alimenta il martelletto di stampa
Eseguire il ritardo di tempo di 3 millisecondi di Printwheel Release
Uscita di PW REL alto

MACROS

Quando si è parlato di sottoprogrammi, si sono fatte note esplicative su una considerazione — il programmatore. I sottoprogrammi hanno un valore addizionale, in quanto se essi possono ridurre il numero delle istruzioni del programma sorgente, allora possono pure ridurre la quantità di tempo spesa nello scrivere il programma sorgente, poichè il tempo di scrittura di un programma sarà direttamente proporzionale alla lunghezza del programma.

Diamo un'altra occhiata al sottoprogramma del ritardo di tempo di due millisecondi. Sebbene in forma di sottoprogramma il programma ha richiesto più byte di codici oggetto, esso non richiede più istruzioni:

Programma vecchio			Programma nuovo		
	LD	A,0FAH		CALL	D2MS
PWSET	DEC	A		-	
	JR	NZ,PWSET		CALL	D2MS
	-			-	
				-	
	-			-	
PWRDY	LD	A,0FAH	D2MS	LD	A,0FAH
RDYDLY	DEC	A	LOPD	DEC	A
	JR	NZ,RDYDLY		JR	NZ,LOPD
				RET	
	6 istruzioni (10 bytes)			6 istruzioni (12 bytes, escludendo le istruzioni di catasta e di inizializzazione	

I sottoprogrammi possono diminuire la lunghezza del vostro programma sorgente, ma aumentare la lunghezza del vostro programma oggetto e il tempo di esecuzione del programma.

I macros diminuiscono la lunghezza del vostro programma sorgente, ma non hanno assolutamente alcun effetto sul vostro programma oggetto.

CHE COSA E' UN MACRO?

DEFINIZIONE DI MACRO

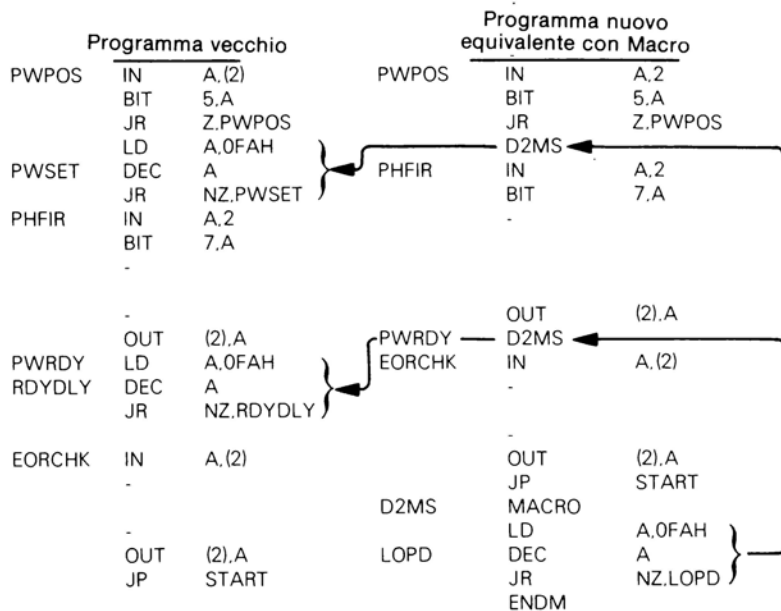
Un macro è una forma di programmazione "stenografica"; essa vi permette di definire una sequenza di istruzioni in una sola forma mnemonica.

Consideriamo la sequenza di istruzioni del ritardo di tempo di due millisecondi; possiamo definirla come una macro, etichettata D2MS, come segue:

```
D2MS    MACRO
        LD      A,0FAH
LOPD    DEC     A
        JR      NZ,LOPD
        ENDM
```

DIRETTIVE MACRO DI ASSEMBLER

Le due istruzioni precedenti ombreggiate sono, in realtà, direttive di assembler: esse raggruppano una sequenza di di istruzioni che da ora in poi possono essere identificate come un gruppo, usando l'etichetta della direttiva assembler MACRO.

Ecco come potremmo usare il ritardo di tempo di due millisecondi nel nostro programma di stampa:

Quando l'Assembler incontra il simbolo D2MS nel campo mnemonico, esso sostituisce questo simbolo con le istruzioni raggruppate dalle direttive MACRO e ENDM. L'Assembler riconosce quale macro usare nel caso che il vostro programma abbia più di una macro, poichè il simbolo nel campo mnemonico deve essere identico all'etichetta di una direttiva MACRO.

Notate che l'Assembler può pure sbrigare alcune faccende associate all'uso di macros. Il "Programma Vecchio" illustrato sopra ha etichette PWSET e RDYDLY per le due istruzioni DEC. Il "Programma Nuovo" ha una sola etichetta, LOPD, nel macro. L'Assembler è abbastanza intelligente da riconoscere quale macro, che compare in una definizione di macro, deve diventare una serie di etichette separate quando si inserisce successivamente il macro un numero di volte nel programma sorgente.

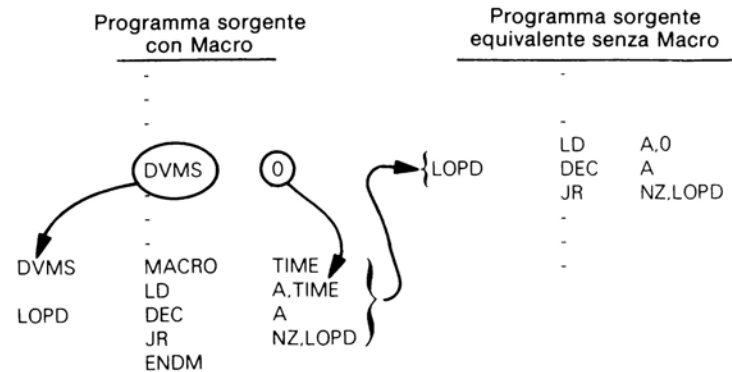
COLLOCAZIONE DELLA DEFINIZIONE DI MACRO IN UN PROGRAMMA SORGENTE

Riepilogando, voi prendete semplicemente una sequenza di istruzioni ripetute, le raggruppate in direttive MACRO e ENDM, quindi date alla direttiva macro un'etichetta unica. Ora usiamo l'etichetta di MACRO come se fosse il mnemonico di un'istruzione. La definizione di macro deve comparire una ed una sola volta, in qualche parte del programma sorgente. E' una buona idea raggruppare tutti i macro ed inserirli all'inizio o alla fine dell'intero programma sorgente.

MACRO CON PARAMETRI

Le istruzioni in un macro possono avere operandi variabili; per esempio possiamo creare un macro di un ritardo di tempo variabile come segue:

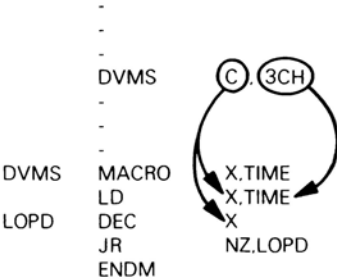
```
DVMS    MACRO   TIME
        LD      A,TIME
LOPD    DEC     A
        JR      NZ,LOPD
        ENDM
```

I simboli che compaiono nel campo dell'operando della direttiva MACRO vengono considerati dall'Assembler come simboli "fittizi"; il riferimento al macro nel corpo del programma sorgente deve includere un equivalente campo d'operando. L'Assembler eguaglierà il campo d'operando di riferimento al macro al campo d'operando della direttiva MACRO, e farà le sostituzioni opportune.

Ecco un altro esempio; il riferimento al macro

```
        DVMS    80H
```

è equivalente a:

```
        LD      A,80H
LOPD    DEC     A
        JR      NZ,LOPD
```

Dipendentemente da quale Assembler usate, potete giocare interessanti partite con la lista dei parametri di macro; in teoria (ma non sempre in pratica), non ci sono restrizioni sulla lunghezza o sulla natura della lista dei parametri del macro. Supponiamo che voi vogliate variare il registro usato nella sequenza di istruzioni del ritardo di tempo; alcuni assembler vi permetteranno di fare come segue:

L'Assembler sostituirà:

```
        DVMS  C,3CH
```

con:

```
        LD    C,3CH
LOPD    DEC   C
        JR    NZ,LOPD
```

Dovrete leggere il manuale dell'Assembler che accompagna il vostro sistema di sviluppo per conoscere le caratteristiche esatte dei macro disponibili.

INTERRUZIONI

Sarebbe arduo giustificare l'inserimento di interruzioni nel sistema a microcalcolatore sviluppato nel Capitolo 4. Infatti le interruzioni dovrebbero essere usate molto avaramente in applicazioni con microcalcolatori.

QUANDO SI USANO LE INTERRUZIONI

Non entreremo in una lunga discussione sulla potenza e sulla debolezza delle interruzioni in sistemi a microcalcolatori; questo soggetto è stato adeguatamente coperto in An Introduction to Microcomputers: Volume I. Per riepilogare, tuttavia, ricordiamo che le interruzioni sono un valido strumento in sistemi a microcalcolatori solo quando si ha a che fare con eventi asincroni e veloci.

Ora, avendo emesso un avvertimento contro l'uso indiscriminato delle interruzioni, procederemo a incorporare un semplice processo d'interruzione nel nostro programma del microcalcolatore per dimostrare come esso funziona.

CONSIDERAZIONI SULL'HARDWARE DELL'INTERRUZIONE

Se si vuole vedere un'interruzione come un processo in un sistema con microcalcolatore Z80 si deve porre un impulso basso nella CPU sull'ingresso del segnale di richiesta di interruzione quando sono state abilitate le interruzioni.

ABILITAZIONE DELLE INTERRUZIONI

Le interruzioni sono abilitate o disabilitate eseguendo rispettivamente le istruzioni EI o DI. Ogni richiesta di interruzione sarà semplicemente ignorata dalla CPU finchè le interruzioni saranno disabilitate.

E' da notare che c'è un'eccezione a questa ultima affermazione: lo Z80 ha un ingresso di interruzione non mascherabile che è sempre abilitato. Questa linea di richiesta di interruzione è usata tipicamente in particolari situazioni quali condizioni di caduta dell'alimentazione e non è rilevante per la nostra discussione.

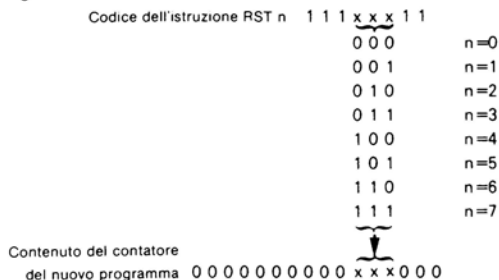
RICONOSCIMENTO DELLE INTERRUZIONI

Se si riceve una richiesta di interruzione mentre le interruzioni sono abilitate, allora dopo il completamento dell'esecuzione della istruzione corrente, la CPU farà uscire un segnale di riconoscimento dell'interruzione (IORQ durante il tempo M1).

La risposta della logica esterna a questo riconoscimento dell'interruzione dipende dal modo con cui si fa funzionare lo Z80 CPU. Ci sono tre modi possibili: 0, 1 e 2.

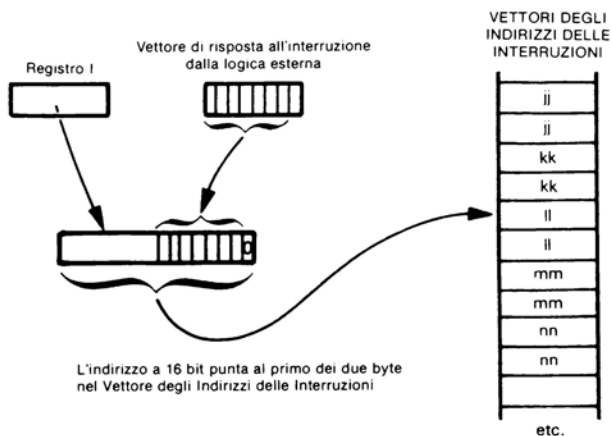
MODO 0 D'INTERRUZIONE DELLA CPU Z80

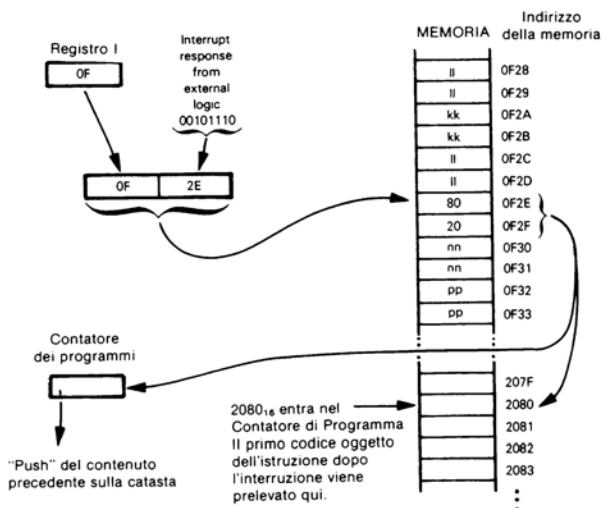
Se la CPU sta funzionando nel Modo 0, ci si aspetta che la logica esterna metta in ingresso un vettore d'interruzione ad otto bit che sarà interpretato come il codice dell'istruzione da eseguire successivamente. Usualmente

si riporterà uno dei possibili otto codici dell'istruzione Restart. Queste istruzioni sono equivalenti a chiamate ad un sottoprogramma di un singolo byte; esse fanno sì che il contenuto del Contatore Programma sia spinto sulla catasta, dopo di che l'esecuzione del programma continua da un indirizzo basso della memoria che può essere calcolato come segue:

MODO 1 D'INTERRUZIONE DELLA CPU Z80

MODO 2 D'INTERRUZIONE DELLA CPU Z80

La logica di risposta alle interruzioni dello Z80 **nel Modo 1 stabilisce automaticamente che la prima istruzione eseguita dopo la risposta all'interruzione sarà un Restart, che salta alla locazione di memoria 0056_{16}.** Se lo Z80 è nel Modo 1 non occorre nessun vettore d'interruzione.

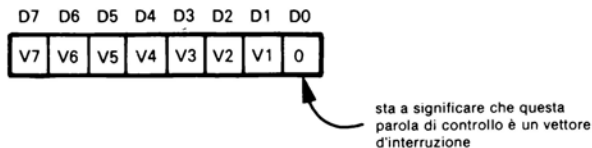
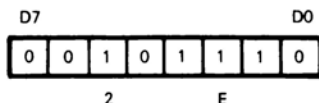
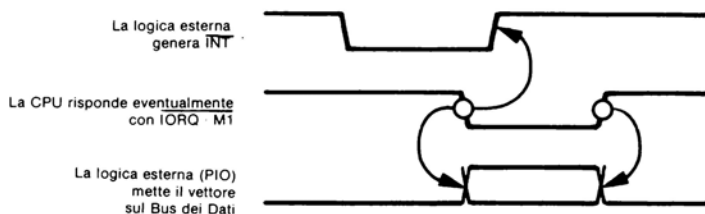
Quando si fa funzionare lo Z80 **nel Modo 2 si deve creare una tabella di vettori di indirizzi delle interruzioni a 16 bit,** che può risiedere dovunque nella memoria indirizzabile. Questi indirizzi a 16 bit identificano la prima istruzione eseguibile dei programmi di servizio delle interruzioni. Quando la CPU riconosce un'interruzione nel Modo 2, **la logica esterna deve porre un vettore di risposta all'interruzione sul Bus dei Dati. La CPU Z80 combinerà il contenuto del registro I col vettore di riconoscimento dell'interruzione per formare un indirizzo a 16 bit, che accederà alla tabella vettoriale degli indirizzi delle interruzioni.** Poichè gli indirizzi a 16 bit si devono trovare anche ai confini degli indirizzi della memoria, si useranno solo sette degli otto bit forniti dalla logica esterna di riconoscimento per creare l'indirizzo della tabella; il bit di ordine minore sarà posto a 0. In tale modo la tabella dei vettori degli indirizzi delle interruzioni a 16 bit sarà indirizzata come segue:

La CPU Z80 eseguirà una Call alla locazione di memoria ottenuta dalla tabella vettoriale degli indirizzi delle interruzioni.

Chiariamo questa logica con un esempio. Supponiamo che ci siano 64 possibili interruzioni esterne; ogni interruzione ha il proprio programma di servizio, quindi si memorizzeranno 64 indirizzi di partenza in 128 byte di memoria. Supponiamo arbitrariamente che questi 128 byte siano memorizzati in una tabella con indirizzi di memoria da $0F00_{16}$ a $0F7F_{16}$. Ora per usare il Modo 2 dovete caricare inizialmente il valore $0F_{16}$ nel registro I dello Z80. Successivamente si riconosce una richiesta esterna di interruzione e la logica esterna di riconoscimento ritorna col vettore $2E_{16}$ sul Bus dei Dati; ecco che cosa succede:

Dalle nostre descrizioni dei modi di interruzione della CPU Z80, è ovvio che il Modo 1 è il più diretto: esso richiede solo che la prima istruzione del nostro programma di interruzione cominci alla locazione di memoria 0056_{16}. Non si richiede nessuna logica esterna per generare un vettore in risposta al riconoscimento dell'interruzione da parte della CPU. **Tuttavia si deve fornire ancora della logica esterna per tastare le condizioni richieste per generare un'interruzione, per generare realmente il segnale di richiesta d'interruzione, e per azzerare il segnale di richiesta d'interruzione una volta che si sia riconosciuta l'interruzione. Tutte queste funzioni possono essere realizzate dal PIO che è già compreso nel nostro sistema indicato in Figura 4-2.** Il solo cambiamento hardware richiesto è il collegamento del segnale INT dal PIO alla CPU, come indicato in Figura 5-1.

Ora dopo aver messo in evidenza la semplicità di funzionamento del Modo 1 della CPU - procederemo trascurando ciò e faremo funzionare la CPU in interruzione nel Modo 2: facciamo ciò perchè il PIO è stato progettato per funzionare specificatamente con la CPU usando la risposta all'interruzione di Modo 2. Come vedremo, questo modo di funzionamento sarà più facile da capire di quanto possa sembrare a prima vista — ciò a causa della logica fornita dal PIO.

Esaminiamo ora come il PIO risponde al riconoscimento dell'interruzione di Modo 2 da parte della CPU.

RISPOSTA DI RICONOSCIMENTO DI UN'INTERRUZIONE DELLO Z80 PIO

Ogni porta (A e B) del PIO ha un vettore d'interruzione indipendente che può essere caricato col valore del vettore desiderato. Il vettore si carica scrivendo una parola di controllo nel registro di controllo della porta nel seguente formato:

D0 è usato come bit di flag che, quando è basso, fa sì che i bit da V7 a V1 siano caricati nel registro Vettore. Nell'istante di riconoscimento di un'interruzione si metterà in ingresso alla CPU il vettore della porta interrompente esattamente nel formato mostrato sopra. Per esempio, facendo riferimento alla nostra discussione sul funzionamento in interruzione della CPU Z80 nel Modo 2, la nostra logica esterna deve fornire un vettore d'interruzione pari a $2E_{16}$. Il formato binario di questo vettore che dovrebbe essere caricato nel registro del PIO è:

Riepilogando, ecco che cosa accade quando la logica esterna (il PIO) richiede una interruzione:

Voi, come progettisti logici o come programmatori, non dovete avere a che fare con la temporizzazione del Bus dei Dati. La combinazione $\overline{IORQ} \cdot \overline{M1}$ è un segnale di riconoscimento dell'interruzione e farà pure correttamente da "strobe" per l'ingresso nella CPU del vettore d'interruzione. Come programmatori, naturalmente, voi dovete avere a che fare con i passi richiesti per porre la CPU e il PIO nei propri modi d'interruzione e di funzionamento e caricare il PIO con il vettore desiderato per il programma di servizio dell'interruzione. Come progettista di sistema, voi dovete pure avere a che fare con la logica richiesta per inizializzare la richiesta d'interruzione. Esamineremo ora questo punto — dopo aver definito ciò, riassumeremo tutte le considerazioni di programmazione risultanti da questo uso delle interruzioni.

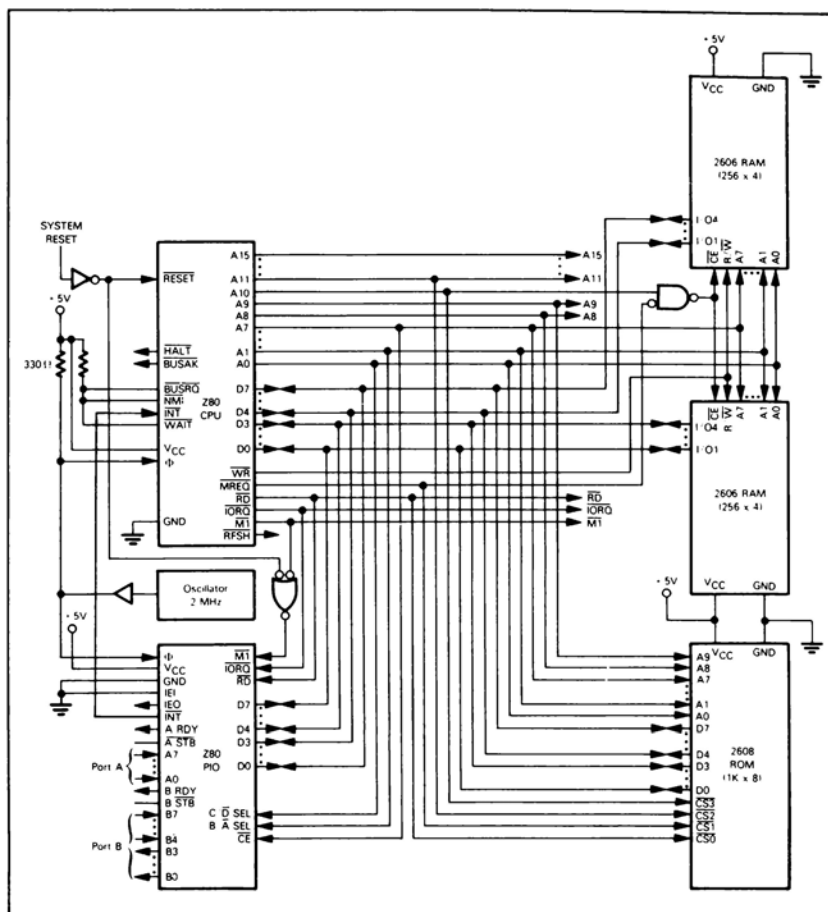
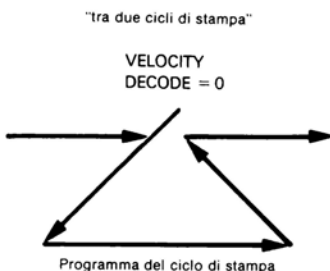



Figura 5-1. Configurazione di un microcalcolatore con lo Z80 facente uso di un PIO per generare un'interruzione

Per determinare che cosa inizializza l'interruzione, **dobbiamo decidere dapprima come useremo l'interruzione.**

Potremmo supporre che il sistema con microcalcolatore sia usato per fare più che una implementazione della logica di un ciclo di stampa. **Supponiamo che ci sia una grande distribuzione della logica della routine di governo richiesta dalla interfaccia della stampante, col risultato che l'intero ciclo di stampa può essere visto come un evento asincrono intermittente. Ora invece di avere un programma che esegue un loop di istruzioni "tra due cicli di stampa", supporremo che si debba eseguire continuamente qualche altro programma tra due cicli di stampa. L'esecuzione del pro-**

gramma del ciclo di stampa ha inizio col segnale VELOCITY DECODE. Ecco il percorso di esecuzione delle istruzioni risultante:

Riferendoci all'assegnazione dei pin del Capitolo 4, vedrete che il segnale VELOCITY DECODE è in ingresso al bit 5 della Porta B dello Z80 PIO. **Per come è stato progettato il PIO, si può usare il segnale VELOCITY DECODE direttamente, senza logica addizionale al di là del PIO, per inizializzare una richiesta di interruzione e dar l'avvio così al programma del ciclo di stampa.**

INIZIALIZZAZIONE DELLE INTERRUZIONI MEDIANTE IL PIO

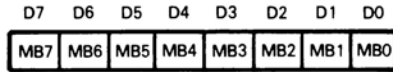
PAROLA DI CONTROLLO DELLE INTERRUZIONI DEL PIO

Il PIO ha una parola di controllo dell'interruzione per ogni porta (A e B) che determina le condizioni sotto le quali una richiesta di interruzione sarà sentita dalla CPU. Nel nostro sistema specificheremo le condizioni d'interruzioni scrivendo una parola nel registro di controllo della Porta B del PIO. La parola di controllo dell'interruzione ha il seguente formato:

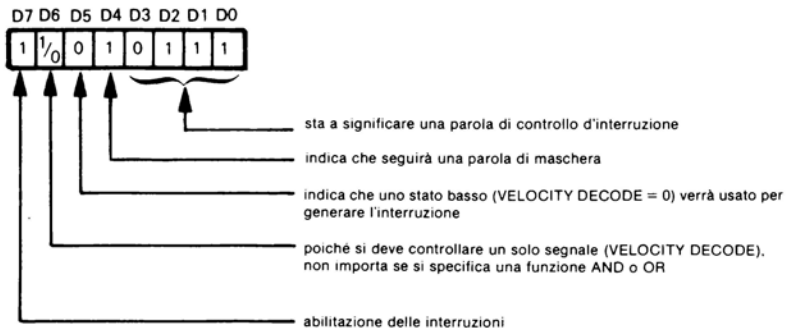
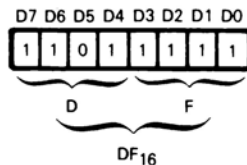
Il bit D7 è usato per abilitare la porta a generare un'interruzione: se il bit 7=1 si possono generare interruzioni. Il bit D6 definisce il funzionamento logico da realizzare per determinare se si deve o no generare una richiesta d'interruzione. Se D6=1 si specifica una funzione AND; tutti i bit della porta selezionati devono andare alti (o bassi, in dipendenza del bit D5) prima che si generi una richiesta d'interruzione. Se il bit D6=0 si specifica allora una funzione OR e si genererà una interruzione se un qualche bit specificato va nello stato attivo (alto o basso).

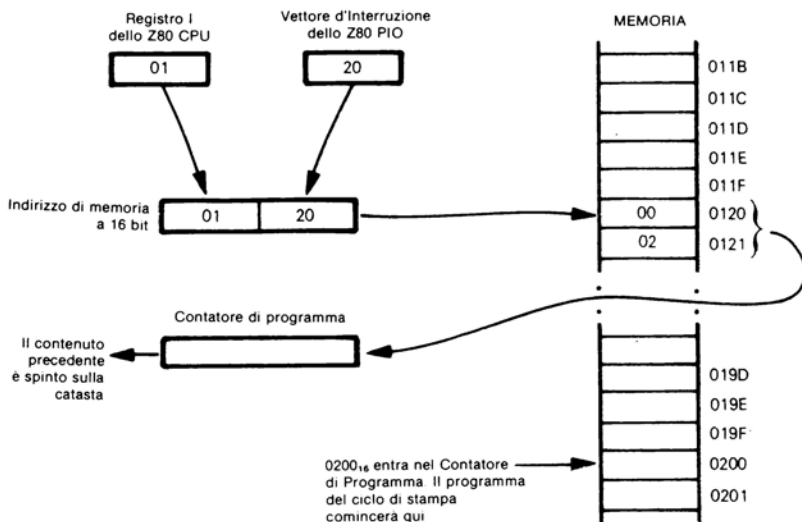
Il bit D5 definisce la polarità attiva della linea del Bus dei Dati che si deve controllare. Se il bit D5=1 si controllano le linee dei dati della porta per uno stato alto; se il bit D5=0, si controllano le linee dei dati per uno stato basso.

Se il bit D4=0, allora si controlleranno tutti i bit in accordo con le regole definite dai bit D6 e D5 della parola di controllo dell'interruzione.

Se D4=1 allora la prossima parola di controllo inviata al PIO deve definire una maschera come segue:

Solo quelle linee della porta il cui bit di maschera è 0 saranno controllate per generare un'interruzione.

Ora, avendo descritto tutte le possibili combinazioni e descrizioni nelle quali il PIO potrebbe generare una richiesta d'interruzione, riferiamo queste possibilità ad un esempio particolare.

Ricordiamo che avevamo a che fare solo col bit 5, che è messo in ingresso alla Porta B del PIO come VELOCITY DECODE. Ora, quando il segnale va basso, vogliamo generare una richiesta d'interruzione per dare l'avvio al programma del ciclo di stampa. **Perciò, la nostra parola di controllo dell'interruzione verso il PIO (Porta B) potrebbe essere questa:**

E la parola di maschera che segue specifica che si controlla solo il bit D5 (VELOCITY DECODE). Il formato della parola di maschera sarebbe:

Ora il solo passo che rimane è il posizionamento del nostro vettore d'interruzione. Se ci riferiamo alla nostra discussione sui funzionamenti della CPU in interruzione nel Modo 2, **vedremo che esso richiede semplicemente che il registro I della CPU e**

il registro del Vettore di Interruzione del PIO siano caricati con valori che saranno combinati per produrre un indirizzo a 16 bit. Dobbiamo pure caricare la locazione specificata da quell'indirizzo e la locazione di memoria adiacente con l'indirizzo della prima istruzione del programma di stampa. Illustriamo nuovamente il funzionamento dell'interruzione nel Modo 2 usando indirizzi arbitrari.

ORIGINE DEL PROGRAMMA D'INTERRUZIONE

Noterete che la locazione reale d'inizio (o origine) specificata dal programma del ciclo di stampa non è importante. Noi non conosciamo quali altri programmi vengono eseguiti nel sistema del microcalcolatore o dove questi altri programmi possano risiedere nella memoria dei programmi, perciò non possiamo assegnare uno spazio di memoria al programma del ciclo di stampa in questo istante. Quando implementerete realmente l'intero sistema del microcalcolatore dovrete fare con molta attenzione una mappa esatta di dove risiedono in memoria tutti i programmi, ma per gli scopi della presente illustrazione questa è una considerazione completamente senza importanza.

Riassumiamo ora i cambiamenti che dobbiamo portare nel nostro programma se vogliamo usare una interruzione per inizializzare il programma del ciclo di stampa. Come vedremo, i cambiamenti sono piuttosto piccoli e la maggior parte di essi consistono in aggiunta di istruzioni per la parte riguardante l'inizializzazione del programma.

```
        ORG  0
; Dapprima facciamo uscire i codici di controllo nel registro di controllo della Porta
; di I/O A
        LD    A,0FFH       ; Posizionamento del Modo 3
        OUT   (1),A
        OUT   (1),A        ; Tutte le linee sono ingressi
; Successivamente facciamo uscire i codici di controllo nel registro di controllo della
; Porta di I/O B
```

```
        OUT  (3),A          ; Posizionamento del Modo 3
        LD   A,0F0H         ; Posizionamento dei pin da 0 a 3 in uscita
        OUT  (3),A          ; ed i pin da 4 a 7 in ingresso
        LD   A,097H         ; Carica la parola di controllo dell'interruzione
        OUT  (3),A
        LD   A,0DFH         ; Posizionamento della parola di maschera della
                            ; interruzione
        OUT  (3),A
        LD   A,020H         ; Carica il vettore d'interruzione (20)
        OUT  (3),A          ; Nel registro vettore della Porta B
; Posizionare quindi la CPU Z80 per il Modo 2 d'interruzione
        IM2                 ; Posizionamento del Modo 2 d'interruzione
        LD   A,010H         ; Carica il registro I della CPU
        LD   I,A            ; Col vettore d'interruzione (01)
        LD   HL,0002H       ; Carica la locazione del vettore d'interruzioni
        LD   (0120H),HL     ; (0120) con l'indirizzo di partenza (0200) del pro-
                            ; gramma del ciclo di stampa
; Posizionare alti HAMMER PULSE, PW READY e PW REL
; Posizionare basso START RIBBON MOTION
        LD   A,7
        OUT  (2),A
; Ora sono state stabilite tutte le condizioni iniziali
; Si possono ora abilitare le interruzioni
        EI
        –
        –
        –
        ORG  0200H
; L'origine della routine di servizio dell'interruzione del programma del ciclo di stam-
; pa è in 0200H, poichè questo è l'indirizzo di esecuzione memorizzato nella loca-
; zione del vettore d'interruzione 0120.
; Programma del ciclo di stampa
; Inizializzazione del ciclo di stampa. Uscita di uno 0 sui bit 0 ed 1 della Porta di I/O
; B e uscita di un 1 sui bit 2 e 3 della Porta di I/O B
START   LD   A,0CH          ; Carica la maschera nell'Accumulatore
        OUT  (2),A          ; Uscita verso la Porta di I/O B
        –
        –
        –
; Alla fine del ciclo di stampa posizionare ad 1 il bit 1 della Porta di I/O B ciò posi-
; ziona alto CH READY
        SET  1,A            ; Posizionamento del bit 1 della Porta B (nell'Ac-
                            ; cumulatore)
        OUT  (2),A          ; Uscita del risultato
        RET
```

Le istruzioni aggiunte al programma illustrato in Figura 4-6 sono ombreggiate, e consistono primariamente in passi necessari per posizionare la CPU e il PIO affinchè funzionino nel modo d'interruzione desiderato. Una volta stabilite tutte le condizioni iniziali, si esegue l'istruzione EI, che abilita la CPU a rispondere alle richieste d'interruzione.

Il programma del ciclo di stampa comincia ora alla locazione di memoria 0200_{16} e sarà inizializzato come risultato di un'interruzione che ha l'avvio da VELOCITY DECODE = 0. E' da notare che le istruzioni "tra due cicli di stampa" all'inizio della

Figura 4-6 sono state tolte; ora è START che identifica la prima istruzione del ciclo di stampa. L'istruzione finale JP START è sostituita dall'istruzione semplice RETURN poichè, in effetti, l'intero programma del ciclo di stampa è chiamato come un sottoprogramma.

SALVATAGGIO DEI REGISTRI E DEGLI STATI

Il metodo che abbiamo appena descritto per vedere una interruzione come un processo è abbastanza semplice: c'è un solo problema — il programma non funzionerà. Abbiamo mostrato un programma di base che è interrotto per eseguire il programma del ciclo di stampa; ma quando sarà interrotto il programma base? Ricordate che il programma che è interrotto è condiviso sia dalla CPU che dai registri del programma del ciclo di stampa. Dobbiamo supporre che il programma interrotto ha utili informazioni memorizzate nei registri, e che forse i flag degli stati hanno un significato che deve essere conservato. Dato il programma di servizio della interruzione illustrato finora, quando torniamo dal programma del ciclo di stampa al programma interrotto, diamo al programma interrotto, in modo arbitrario, il contenuto dei registri col quale finisce il programma del ciclo di stampa. Ciò non si deve fare mai. **Dobbiamo perciò sostenere il programma di esecuzione del ciclo di stampa con istruzioni che salvano il contenuto dei registri e degli stati — prima di modificare un solo registro o stato; alla fine del programma, si devono rimemorizzare i contenuti originali dei registri e degli stati.** Tipicamente i contenuti dei registri e degli stati sono salvati spingendoli sulla catasta, e rimemorizzandoli alla fine del programma, tirandoli fuori dalla catasta. La sequenza delle istruzioni potrebbe essere come segue:

```
          ORG    0200H
; Origine della routine di servizio di interruzione del programma del ciclo di stampa
; a 0200H, poichè questo è l'indirizzo di esecuzione memorizzato alla locazione del
; vettore d'interruzione 0120.
START     PUSH   AF         ; Salvataggio dell'Accumulatore e dei flag
          PUSH   BC         ; Salvataggio dei registri B e C
          PUSH   DE         ; Salvataggio dei registri D ed E
          PUSH   HL         ; Salvataggio dei registri H ed L
; Inizializzazione del ciclo di stampa — uscita di uno 0 sui bit 0 ed 1 della Porta di
; I/O B ed uscita di un 1 sui bit 2 e 3 della Porta di I/O B
          LD     A,0CH      ; Carica la maschera nell'Accumulatore
          OUT    (2),A      ; Uscita verso la Porta di I/O B
          —
          —
          —
; Alla fine del ciclo di stampa posizionare ad 1 il bit 1 della Porta di I/O B . Ciò
; posiziona alto CH RDY
          SET    1,A        ; Posizionamento del bit 1 della Porta di I/O B (nel-
                            ; l'Accumulatore)
          OUT    (2),A      ; Uscita del risultato
          POP    HL         ; Ripristino dei registri H ed L
          POP    DE         ; Ripristino dei registri D ed E
          POP    BC         ; Ripristino dei registri B e C
          POP    AF         ; Ripristino dell'Accumulatore e dei flag
          RET
```

L'intera sequenza di salvataggio e di ripristino aggiunge al nostro programma un totale di otto istruzioni. Dal momento che voi ricordate di fare il "pop" del contenuto dei registri e degli stati in ordine inverso rispetto a quello con cui avete fatto il "push", voi non avrete nessun problema.

USO DEI REGISTRI AUSILIARI DELLA CPU Z80

Come abbiamo stabilito all'inizio di questa discussione, la sequenza push/pop è il tipico metodo usato per salvare/ripristinare il contenuto dei registri e degli stati. La CPU Z80, tuttavia, fornisce una caratteristica architettonica molto atipica che può essere usata per semplificare il processo di salvataggio/ripristino. Ricorderete che la CPU Z80 fornisce due insiemi di registri a scopi generali accoppiati come indicato sotto.

	F	Parole degli stati di programma		F'
	A	Accumulatori Primari		A'
B	C	Accumulatori Secondari/Contatori di Dati	B'	C'
D	E	Accumulatori Secondari/Contatori di Dati	D'	E'
H	L	Accumulatori Secondari/Contatori di Dati	H'	L'
SP		Puntatore della Catasta		
PC		Contatore di Programma		
IX		Registro Indice X		
IY		Registro Indice Y		
	IV	Vettore d'interruzione		
	R	Contatore di Rinfresco della Memoria		

Ora l'insieme di istruzioni dello Z80 contiene due istruzioni che permettono lo scambio dei contenuti di questi insiemi di registri duplicati. L'istruzione EX AF,AF' scambia il contenuto dei registri A ed F col contenuto dei registri A' ed F'. L'istruzione EXX scambia il contenuto delle coppie di registri BC, DE e HL rispettivamente con il contenuto delle coppie di registri B'C', D'E' e H'L'. **Perciò la sequenza delle quattro istruzioni PUSH per salvare i registri e delle quattro istruzioni POP per ripristinare i registri possono essere sostituite usando le istruzioni EX AF,AF' ed EXX come segue:**

```
Programma vecchio               Programma nuovo
START   PUSH   AF               START   EX AF,AF'
        PUSH   BC                       EXX
        PUSH   DE                       -
        PUSH   HL                       -
        -                               EX AF,AF'
                                        EXX
                                        RET
        -
        POP    HL
        POP    DE
        POP    BC
        POP    AF
        RET
```

L'uso delle istruzioni Exchange invece delle istruzioni PUSH/POP ha salvato in totale quattro istruzioni e inoltre si ha il risultato di una risposta molto più veloce ad una interruzione, poichè l'esecuzione delle due istruzioni Exchange richiede solo un quinto del tempo richiesto per eseguire le quattro istruzioni PUSH. **Un altro vantag-**

gio delle istruzioni Exchange è che non si è usata nessuna memoria a lettura e scrittura per questa sequenza, mentre la sequenza PUSH/POP usa otto byte della memoria di catasta.

Naturalmente le istruzioni Exchange si possono usare solo per un solo livello di interruzioni; se esistono livelli multipli di interruzioni, si devono servire interruzioni annidate, quindi si deve usare la catasta per salvare il contenuto dei registri. Vediamo ora che cosa altro richiedono le interruzioni multiple.

INTERRUZIONI MULTIPLE

Che cosa succede se il vostro sistema a microcalcolatore è collegato a più di un dispositivo di logica esterna che sia capace di richiedere interruzioni? Per esempio, un solo sistema con microcalcolatore Z80 potrebbe pilotare un certo numero di stampanti. Senza entrare nell'economia delle configurazioni delle interruzioni multiple del microcalcolatore, esaminiamo i modi con cui si possono maneggiare le interruzioni multiple.

La sola cosa che cambia quando si passa da interruzioni singole a interruzioni multiple è il fatto che il programma di servizio dell'interruzione non è più unico. **Ci deve essere un programma di servizio dell'interruzione diverso per ogni dispositivo esterno capace di richiedere un'interruzione.** Ciò a sua volta significa conoscere quale programma di servizio dell'interruzione si deve eseguire. Inoltre, **se più di un dispositivo richiede contemporaneamente servizio di interruzione, quale riconosceremo — e in quale ordine?** Questi sono problemi di vettorizzazione delle interruzioni e di arbitrarietà delle priorità, argomenti trattati in dettaglio in An Introduction to Microcomputers: Volume I — Basic Concepts. Non ripeteremo in questo libro la discussione di questi concetti di base; piuttosto, vedremo i modi pratici con i quali si possono servire interruzioni multiple in un sistema a microcalcolatore con lo Z80. Vedremo che il progetto con la CPU Z80 e con lo Z80 PIO rende il servizio delle interruzioni multiple molto diretto.

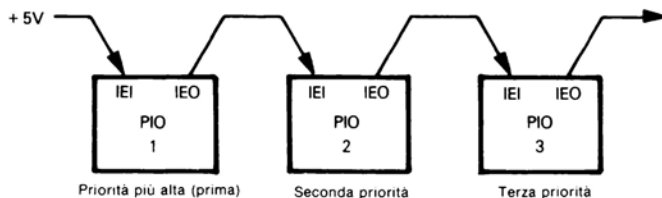
Ci sono innumerevoli modi di implementazione delle interruzioni multiple in un sistema col microcalcolatore Z80 ed è certamente al di là dello scopo di questo libro esplorarle tutte. Perciò **limiteremo la nostra discussione al metodo più ovvio e più diretto — il metodo che ha come supporto il progetto della CPU Z80 e dello Z80 PIO** (come pure altri dispositivi dello Z80 che non abbiamo potuto descrivere in questo libro).

Come abbiamo appena stabilito, **i due problemi principali che si devono risolvere in sistemi utilizzanti interruzioni multiple sono: 1) vettorizzazione delle interruzioni e 2) arbitrarietà delle priorità.**

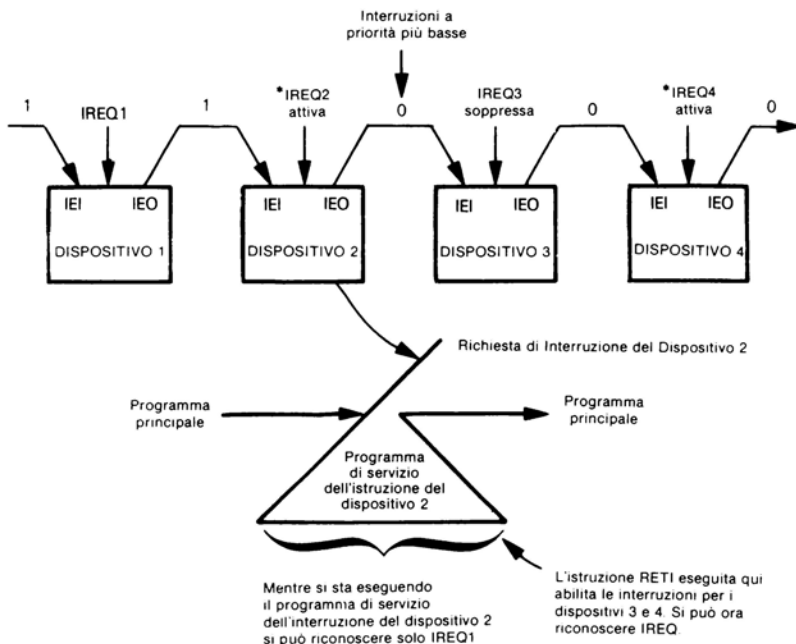
La vettorizzazione delle interruzioni è già stata descritta precedentemente in questo capitolo quando si è discusso **del funzionamento d'interruzione della CPU Z80 nel Modo 2.** Questo modo di funzionamento **permette la vettorizzazione di un numero quasi illimitato di dispositivi interrompenti.** L'unico requisito posto al dispositivo interrompente è che esso deve rispondere al riconoscimento dell'interruzione da parte della CPU mettendo un vettore a 7 bit sul Bus dei Dati del sistema. Ciò è realizzato automaticamente dallo Z80 PIO, ma potrebbe essere ottenuto molto facilmente dalla logica del vostro stesso progetto.

ARBITRARIETA' SULLA PRIORITA' DELLE INTERRUZIONI

L'arbitrarietà della priorità delle interruzioni è fornita pure dallo Z80 PIO, e una discussione su come questo dispositivo realizza l'arbitrarietà servirà pure come esempio della teoria generale coinvolta. Lo Z80 PIO usa un tipico schema a "daisy chain" per posizionare le priorità delle interruzioni. Interrupt Enable In (IEI) e Interrupt Enable Out (IEO) sono

segnali standard della priorità delle interruzioni a "daisy chain". Quando nel sistema è presente più di un PIO, il PIO a priorità più alta (cioè quello elettricamente più vicino alla CPU) avrà IEI collegato a +5V e sarà collegato col suo IEO all'IEI del PIO della "daisy chain" avente la priorità successiva più alta:

Il modo di ottenere una "daisy chain" è stato descritto ben dettagliatamente nel Volume I. Se dopo aver letto questo paragrafo voi non siete sicuri sulla rete di priorità del "daisy chain", fate riferimento al Volume I per chiarimenti. Quando più di un dispositivo richiede interruzione, un riconoscimento si muove lungo la "daisy chain" finchè non viene catturato dal dispositivo richiedente l'interruzione elettricamente più vicino alla CPU. Non appena il processo di riconoscimento dell'interruzione è finito, si esegue un programma di servizio dell'interruzione riconosciuta; la logica esterna di riconoscimento toglierà ora la sua richiesta di interruzione. Nella gran parte dei sistemi a microcalcolatore, senza che la CPU disabiliti ulteriori interruzioni, il dispositivo a priorità minore può interrompere immediatamente il programma di servizio dell'interruzione di un dispositivo a maggiore priorità. Non è il caso di un sistema con lo Z80. Un dispositivo la cui richiesta di interruzione è stata riconosciuta continua a sopprimere le richieste di interruzioni dei dispositivi a minore priorità nella "daisy chain", finchè non si rilevi sul Bus dei Dati il secondo byte del codice oggetto di un'istruzione RETI o RETN. Il dispositivo riconosciuto risponde ad un codice oggetto di un'istruzione RETI o RETN riabilitando le interruzioni per i dispositivi a minore priorità nella "daisy chain".

Stabilendo che il sistema col microcalcolatore Z80 è stato progettato per fare un uso corretto delle istruzioni RETI o RETN, la logica di arbitrarietà della priorità delle interruzioni permetterà che un programma di servizio di un interruzione sia interrotto solo da una richiesta d'interruzione a priorità maggiore.

Una illustrazione dello schema dell'arbitrarietà delle priorità delle interruzioni dello Z80 si può vedere in figura a pagina 5-38.

GIUSTIFICAZIONE DELLE INTERRUZIONI

I programmatori di minicalcolatori e i programmatori di grandi calcolatori fanno un uso indiscriminato delle interruzioni e semplicemente per suddividere il costo della CPU (Central Processing Unit) tra un numero di applicazioni diverse.

ECONOMICITA' DELLE INTERRUZIONI

Voi, come utilizzatori di microcalcolatori, dovete giustificare la suddivisione di un costo che può essere compreso tra 5 e 20 dollari. A fronte di questo costo dovete caricare il costo della logica esterna necessaria per creare i segnali di richiesta di interruzione — come pure i costi extra di programmazione. **Le specifiche economiche rendono ovvia la necessità vitale di interruzioni in sistemi con microcalcolatore.** Dovete esaminare la vostra applicazione con attenzione prima di stabilire che le interruzioni rappresentano la soluzione da seguire. Una seconda CPU o un secondo sistema con microcalcolatore, saranno frequentemente più economici che usare interruzioni per suddividere un solo sistema a microcalcolatore tra un numero di applicazioni diverse.

CONSIDERAZIONI SULLA TEMPORIZZAZIONE DELLE INTERRUZIONI

Supposto che le interruzioni per la vostra applicazione siano economiche, sono pure importanti considerazioni sulla temporizzazione.

Certamente le interruzioni sembrano molto attrattive quando la vostra applicazione maneggia eventi asincroni. Nel vostro caso, **supponiamo che il ciclo di stampa medio duri approssimativamente 10 millisecondi; supponiamo inoltre che sia impossibile dire se l'intervallo di tempo tra due cicli di stampa sarà 1 o 100 millisecondi. In questo caso, per eseguire qualche altro programma nell'intervallo di tempo tra due cicli di stampa, dobbiamo usare interruzioni per inizializzare il ciclo di stampa —** poichè non abbiamo alcuna idea di quando inizi il successivo ciclo di stampa.

In realtà il tempo che intercorre tra due cicli di stampa sarà noto molto accuratamente. **Una stampante avrà una velocità di stampa dei caratteri nota.** Se questa velocità è di 45 caratteri al secondo, allora si richiederanno 22,2 millisecondi per stampare un carattere. Se dei 22 millisecondi dieci sono necessari per eseguire realmente il programma del ciclo di stampa, allora rimarranno 12 millisecondi tra due cicli di stampa. **Non abbiamo più bisogno di interruzioni.** Non appena il programma che si esegue tra due cicli di stampa è diviso in segmenti, ognuno dei quali si esegue in 12 millisecondi o meno, allora ogni segmento può terminare con un loop di istruzioni che fa un test sullo stato dell'ingresso di "velocity decode" per inizializzare il successivo ciclo di stampa:

```
START:  IN   A,(2)      ; La Porta di I/O B entra nell'Accumulatore
        BIT  5,A        ; Test sul bit 5
        JR   NZ,START   ; Se non è zero, ritornare a START
```

Capitolo 6

L'INSIEME DI ISTRUZIONI DELLO Z80

Le istruzioni spaventano falsamente gli utilizzatori di microcalcolatori che sono nuovi alla programmazione. Prese come eventi isolati, le operazioni associate all'esecuzione di una singola istruzione sono abbastanza facili da seguire — e ciò è lo scopo di questo capitolo.

Perchè le istruzioni di un microcalcolatore si riferiscono ad un "insieme di istruzioni"? La risposta è che le istruzioni scelte dai progettisti di un qualsiasi microcalcolatore sono scelte con grande cura; deve essere facile eseguire operazioni complesse come una sequenza di semplici eventi — ognuno di questi è rappresentato da una sola istruzione di un "insieme" di istruzioni ben progettato.

Rimanendo fedeli a An Introduction to Microcomputers, Volume II, la Tabella 6-1 riassume l'insieme di istruzioni del microcalcolatore Z80, raggruppando insieme le istruzioni analoghe.

Le singole istruzioni sono descritte successivamente in ordine alfabetico della mnemonica delle istruzioni.

Inoltre per semplificare la dichiarazione che fa ogni istruzione, si identifica l'obiettivo dell'istruzione in una normale logica di programmazione.

ABBREVIAZIONI

Ecco le abbreviazioni usate in questo capitolo:

A, F, B, C, D, E, H, L	I registri ad 8 bit. A è l'Accumulatore ed F è la Parola dello Stato del Programma.
AF', BC', DE', HL'	Le coppie alternative di registri
addr	Un indirizzo di memoria a 16 bit
x(b)	Bit b di un registro a 8 bit o di una locazione di memoria x
cond	Condizione per un salto di programma. Le condizioni sono:

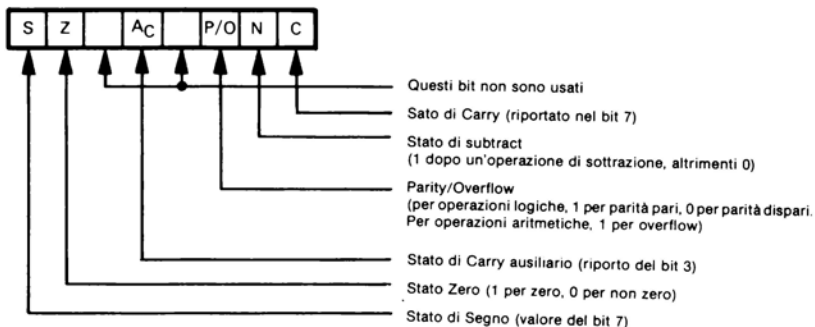
NZ	—	Non Zero (Z=0)	
Z	—	Zero (Z=1)	
NC	—	Non carry (C=0)	(Nessun Riporto)
C	—	Carry (C=1)	(Riporto)
PO	—	Parity Odd (P=0)	(Parità dispari)
PE	—	Parity Even (P=1)	(Parità pari)
P	—	Sign Positive (S=0)	(Segno Positivo)
M	—	Sign Negative (S=1)	(Segno Negativo)

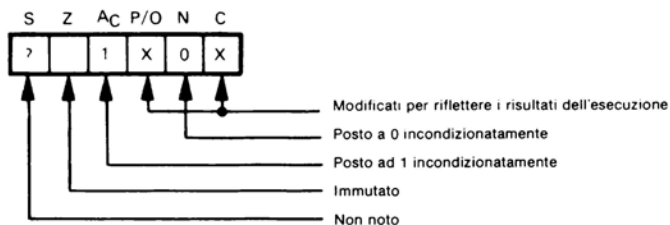
data	Un'unità dati binari a 8 bit
data 16	Un'unità dati binari a 16 bit
disp	Un dislocamento di un indirizzo binario a 8 bit con segno
xx(H)	Gli 8 bit di ordine maggiore di una quantità xx a 16 bit

IV	Registro del vettore d'interruzione (8 bit)
IX,IY	I registri Indice (ciascuno a 16 bit)
xy	L'uno o l'altro dei registri Indice (IX o IY)
LSB	Least Significant Bit (Bit 0) (Bit Meno Significativo)
Label	Un indirizzo a 16 bit di un'istruzione di memoria
xx(LO)	Gli 8 bit di ordine minore di una quantità xx a 16 bit
MSB	Most Significant Bit (Bit 7) (Bit più Significativo)
PC	Program Counter (Contatore di Programma)
port	Un indirizzo di 8 bit di una porta di I/O
pr	Una qualsiasi delle seguenti coppie di registri: BC DE HL AF
R	Il registro di Refresh (8 bit)
reg	Uno qualsiasi dei seguenti registri: A B C D E H L
rp	Una qualsiasi delle seguenti coppie di registri: BC DE HL SP
SP	Stack Pointer (16 bit) (Puntatore della Catasta)
Statuses	Lo Z80 ha i seguenti flag di stato: C — Stato di Carry Z — Stato di Zero S — Stato di Segno P/O — Stato di Parità/Overflow A_C — Stato di Carry ausiliario N — Stato di Subtract Nelle colonne dello stato si usano i seguenti simboli: X — Il flag è influenzato dall'operazione (blank) — Il flag non è influenzato dall'operazione 1 — Il flag è posto a 1 dall'operazione 0 — Il flag è posto a 0 dall'operazione ? — Il flag è sconosciuto dopo l'operazione P — Il flag mostra lo stato di parità O — Il flag mostra lo stato di overflow I — Il flag mostra lo stato di interruzione abilitata/disabilitata
[ ]	Contenuto della locazione racchiusa in parentesi. Se la designazione di un registro è racchiusa tra parentesi, allora si specifica il contenuto del registro indicato. Se si racchiude tra parentesi un numero di una porta di I/O, allora si specifica il contenuto della porta di I/O. Se si racchiude tra

	parentesi un indirizzo di memoria, allora si specifica la locazione di memoria indirizzata.
[[ ]]	Indirizzamento implicato di memoria; il contenuto della locazione di memoria indicata dal contenuto di un registro.
$\wedge$	AND logico
$\vee$	OR logico
$\forall$	Esclusiv-OR logico
$\leftarrow$	Il dato è trasferito nella direzione della freccia
$\leftarrow \rightarrow$	Il dato è scambiato tra le due locazioni indicate sui due lati della freccia

STATO

I sei flag di stato sono memorizzati in un registro di Flag (F) come segue:

F ed A sono trattati qualche volta come una coppia di registri.

L'effetto dell'esecuzione di un'istruzione sullo stato è illustrato come segue:

CAMBIAMENTI DI STATO CON L'ESECUZIONE DI UN'ISTRUZIONE

Nelle illustrazioni dell'esecuzione di un'istruzione una X identifica uno stato che è posto a 1 o a 0. Uno 0 identifica uno stato che è sempre azzerato. Un 1 identifica uno stato che è sempre posto ad 1. Uno spazio (blank) significa che lo stato non cambia. Un punto interrogativo (?) significa che lo stato è ignoto.

MNEMONICA DELLE ISTRUZIONI

La parte fissa di un'istruzione in linguaggio assembly è mostrata con LETTERE MAIUSCOLE.

La parte variabile (dato immediato, numero del dispositivo di I/O, nome del registro, etichetta o indirizzo) è mostrato con lettere minuscole.

CODICI OGGETTO DELLE ISTRUZIONI

I codici oggetto delle istruzioni sono rappresentati come due pesi esadecimali per istruzioni senza variazioni.

I codici oggetto delle istruzioni sono rappresentati come otto pesi binari per istruzioni con variazioni; sono perciò identificabili le variazioni delle rappresentazioni con pesi binari.

CODICI E TEMPI DI ESECUZIONE DELLE ISTRUZIONI

La Tabella 6-2 elenca le istruzioni in ordine alfabetico, mostrando i codici oggetto e i tempi di esecuzione espressi in cicli macchina.

Dove sono indicati due cicli di istruzioni, il primo si riferisce al caso di "condizione non verificata" mentre il secondo si riferisce al caso di "condizione verificata".

*Address Bus: A0-A7: [C]
A8-A15: [B]

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
O/I	IN	A,port	2							[A]←[port] Input to Accumulator from directly addressed I/O port. Address Bus: A0-A7: port A8-A15: [A]
	IN	reg,(C)	2		X	X	P	X	0	[reg]←[[C]] Input to register from I/O port addressed by the contents of C.* If second byte is 70_{16} only the flags will be affected.
	INIR		2		1	?	?	?	1	Repeat until [B]=0: [[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a block of data from I/O port addressed by contents of C to memory location addressed by contents of HL, going from low addresses to high. Contents of B serve as a count of bytes remaining to be transferred.*
	INDR		2		1	?	?	?	1	Repeat until [B]=0: [[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a block of data from I/O port addressed by contents of C to memory location addressed by contents of HL, going from high addresses to low. Contents of B serve as a count of bytes remaining to be transferred.*
	INI		2		X	?	?	?	1	[[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a byte of data from I/O port addressed by contents of C to memory location addressed bv contents of HL. Decrement byte count and increment destination address.*

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

*Address Bus: A0-A7: [C]
A8-A15: [B]

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
I/O (Continua)	IND		2		X	?	?	?	1	[[HL]]←[[C]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a byte of data from I/O port addressed by contents of C to memory location addressed by contents of HL. Decrement both byte count and destination address.*
	OUT	port,A	2							[port]←[A] Output from Accumulator to directly addressed I/O port. Address Bus: A0-A7: port A8-A15: [A]
	OUT	(C),reg	2							[[C]]←[reg] Output from register to I/O port addressed by the contents of C.*
	OTIR		2		1	?	?	?	1	Repeat until [B]=0: [[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a block of data from memory location addressed by contents of HL to I/O port addressed by contents of C, going from low memory to high. Contents of B serve as a count of bytes remaining to be transferred.*
	OTDR		2		1	?	?	?	1	Repeat until [B]=0: [[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a block of data from memory location addressed by contents of HL to I/O port addressed by contents of C, going from high memory to low. Contents of B serve as a count of bytes remaining to be transferred.*

*Address Bus: A0-A7: [C]
A8-A15: [B]

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO C	STATO Z	STATO S	STATO P/O	STATO A_C	STATO N	OPERAZIONE EFFETTUATA
I/O (Continua)	OUTI		2		X	?	?	?	1	[[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] + 1 Transfer a byte of data from memory location addressed by contents of HL to I/O port addressed by contents of C. Decrement byte count and increment source address.*
	OUTD		2		X	?	?	?	1	[[C]]←[[HL]] [B]←[B] - 1 [HL]←[HL] - 1 Transfer a byte of data from memory location addressed by contents of HL to I/O port addressed by contents of C. Decrement both byte count and source address.*
RIFERIMENTO ALLA MEMORIA PRIMARIO	LD	A,(addr)	3							[A]←[addr] Load Accumulator from directly addressed memory location.
	LD	HL,(addr)	3							[H]←[addr + 1], [L]←[addr] Load HL from directly addressed memory.
	LD	rp,(addr) xy,(addr)	4							[rp(HI)]←[addr + 1], [rp(LO)]←[addr] or [xy(HI)]←[addr + 1], [xy(LO)]←[addr] Load register pair or Index register from directly addressed memory.
	LD	(addr),A	3							[addr]←[A] Store Accumulator contents in directly addressed memory location.
	LD	(addr),HL	3							[addr + 1]←[H], [addr]←[L] Store contents of HL to directly addressed memory location.
	LD	(addr),rp (addr),xy	4							[addr + 1]←[rp(HI)], [addr]←[rp(LO)] or [addr + 1]←[xy(HI)], [addr]←[xy(LO)] Store contents of register pair or Index register to directly addressed memory.
	LD	A,(BC) A,(DE)	1							[A]←[[BC]] or [A]←[[DE]] Load Accumulator from memory location addressed by the contents of the specified register pair.

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO C	STATO Z	STATO S	STATO P/O	STATO A_C	STATO N	OPERAZIONE EFFETTUATA
RIFERIMENTO ALLA MEMORIA PRIMARIO (Continua)	LD	reg,(HL)	1							[reg]←[[HL]] Load register from memory location addressed by contents of HL.
	LD	(BC),A (DE),A	1							[[BC]]←[A] or [[DE]]←[A] Store Accumulator to memory location addressed by the contents of the specified register pair.
	LD	(HL),reg	1							[[HL]]←[reg] Store register contents to memory location addressed by the contents of HL.
	LD	reg,(xy + disp)	3							[reg]←[[xy] + disp] Load register from memory location using base relative addressing.
	LD	(xy + disp),reg	3							[[xy] + disp]←[reg] Store register to memory location addressed relative to contents of Index register.
RICERCA E TRASFERIMENTO DI UN BLOCCO	LDIR		2				0	0	0	Repeat until [BC]=0: [[DE]]←[[HL]] [DE]←[DE] + 1 [HL]←[HL] + 1 [BC]←[BC] - 1 Transfer a block of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE, going from low addresses to high. Contents of BC serve as a count of bytes to be transferred.
	LDDR		2				0	0	0	Repeat until [BC]=0: [[DE]]←[[HL]] [DE]←[DE] - 1 [HL]←[HL] - 1 [BC]←[BC] - 1 Transfer a block of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE, going from high addresses to low. Contents of BC serve as a count of bytes to be transferred.

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICO	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
RICERCA E TRASFERIMENTO DI UN BLOCCO (Continua)	LDI		2				X	0	0	[[DE]] ← [[HL]] [DE] ← [DE] + 1 [HL] ← [HL] + 1 [BC] ← [BC] - 1 Transfer one byte of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE. Increment source and destination addresses and decrement byte count.
	LDD		2				X	0	0	[[DE]] ← [[HL]] [DE] ← [DE] - 1 [HL] ← [HL] - 1 [BC] ← [BC] - 1 Transfer one byte of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE. Decrement source and destination addresses and byte count.
	CPIR		2		X	X	X	X	1	Repeat until [A]=[[HL]] or [BC]=0: [A] - [[HL]] (only flags are affected) [HL] ← [HL] + 1 [BC] ← [BC] - 1 Compare contents of Accumulator with those of memory block addressed by contents of HL, going from low addresses to high. Stop when a match is found or when the byte count becomes zero.
	CPDR		2		X	X	X	X	1	Repeat until [A]=[[HL]] or [BC]=0: [A] - [[HL]] (only flags are affected) [HL] ← [HL] - 1 [BC] ← [BC] - 1 Compare contents of Accumulator with those of memory block addressed by contents of HL, going from high addresses to low. Stop when a match is found or when the byte count becomes zero.

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
RICERCA E TRASFERIMENTO DI UN BLOCCO (Continua)	CPI		2		X	X	X	X	1	[A] - [[HL]] (only flags are affected) [HL]←[HL] + 1 [BC]←[BC] - 1 Compare contents of Accumulator with those of memory location addressed by contents of HL. Increment address and decrement byte count.
	CPD		2		X	X	X	X	1	[A] - [[HL]] (only flags are affected) [HL]←[HL] - 1 [BC]←[BC] - 1 Compare contents of Accumulator with those of memory location addressed by contents of HL. Decrement address and byte count.
RIFERIMENTO ALLA MEMORIA SECONDARIO	ADD	(HL) (xy + disp)	1 3	X	X	X	O	X	0	[A]←[A] + [[HL]] or [A]←[A] + [[xy] + disp] Add to Accumulator using implied addressing or base relative addressing.
	ADC	(HL) (xy + disp)	1 3	X	X	X	O	X	0	[A]←[A] + [[HL]] + C or [A]←[A] + [[xy] + disp] + C Add with Carry using implied addressing or base relative addressing.
	SUB	(HL) (xy + disp)	1 3	X	X	X	O	X	1	[A]←[A] - [[HL]] or [A]←[A] - [[xy] + disp] Subtract from Accumulator using implied addressing or base relative addressing.
	SBC	(HL) (xy + disp)	1 3	X	X	X	O	X	1	[A]←[A] - [[HL] - C or [A]←[A] - [[xy] + disp] - C Subtract with Carry using implied addressing or base relative addressing.
	AND	(HL) (xy + disp)	1 3	0	X	X	P	1	0	[A]←[A] Λ [[HL]] or [A]←[A] Λ [[xy] + disp] AND with Accumulator using implied addressing or base relative addressing
	OR	(HL) (xy + disp)	1 3	0	X	X	P	1	0	[A]←[A] V [[HL]] or [A]←[A] V [[xy] + disp] OR with Accumulator using implied addressing or base relative addressing.
	XOR	(HL) (xy + disp)	1 3	0	X	X	P	1	0	[A]←[A] ⊻ [[HL]] or [A]←[A] ⊻ [[xy] + disp] Exclusive-OR with Accumulator using implied addressing or base relative addressing.
	CP	(HL) (xy + disp)	1 3	X	X	X	O	X	1	[A] - [[HL]] or [A] - [[xy] + disp] Compare with Accumulator using implied addressing or base relative addressing. Only the flags are affected.
	INC	(HL) (xy + disp)	1 3		X	X	O	X	0	[[HL]]←[[HL]] + 1 or [[xy] + disp]←[[xy] + disp] + 1 Increment using implied addressing or base relative addressing.
	DEC	(HL) (xy + disp)	1 3		X	X	O	X	1	[[HL]]←[[HL]] - 1 or [[xy] + disp]←[[xy] + disp] - 1 Decrement using implied addressing or base relative addressing.

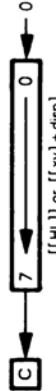
Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
ROTAZIONE E SPOSTAMENTO DI MEMORIA	RLC	(HL) (xy + disp)	2 4	X	X	X	P	0	0	[[HL]] or [[xy] + disp] Rotate contents of memory location (implied or base relative addressing) left with branch Carry.
	RL	(HL) (xy + disp)	2 4	X	X	X	P	0	0	[[HL]] or [[xy] + disp] Rotate contents of memory location left through Carry.
	RRC	(HL) (xy + disp)	2 4	X	X	X	P	0	0	[[HL]] or [[xy] + disp] Rotate contents of memory location right with branch Carry.
	RR	(HL) (xy + disp)	2 4	X	X	X	P	0	0	[[HL]] or [[xy] + disp] Rotate contents of memory location right through Carry.
	SLA	(HL) (xy + disp)	2 4	X	X	X	P	0	0	[[HL]] or [[xy] + disp] Shift contents of memory location left and clear LSB (Arithmetic Shift).
	SRA	(HL) (xy + disp)	2 4	X	X	X	P	0	0	[[HL]] or [[xy] + disp] Shift contents of memory location right and preserve MSB (Arithmetic Shift).
	SRL	(HL) (xy + disp)	2 4	X	X	X	P	0	0	[[HL]] or [[xy] + disp] Shift contents of memory location right and clear MSB (Logical Shift).

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO C	STATO Z	STATO S	STATO P/O	STATO A_C	STATO N	OPERAZIONE EFFETTUATA
IMMEDIATO	LD	reg.data	2							[reg] ← data Load immediate into register.
	LD	rp.data16 xy.data16	3 4							[rp] ← data16 or [xy] ← data16 Load 16 bits of immediate data into register pair or Index register
	LD	(HL).data (xy + disp).data	2 4							[[HL]] ← data or [[xy] + disp] ← data Load immediate into memory location using implied or base relative addressing
SALTO	JP	label	3							[PC] ← label Jump to instruction at address represented by label
	JR	disp	2							[PC] ← [PC] + 2 + disp Jump relative to present contents of Program Counter
	JP	(HL) (xy)	1 2							[PC] ← [HL] or [PC] ← [xy] Jump to address contained in HL or Index register.
RITORNO DA E CHIAMATA DI UN SOTTOPROGRAMMA	CALL	label	3							[[SP]-1] ← [PC(HI)] [[SP]-2] ← [PC(LO)] [SP] ← [SP] 2 [PC] ← label Jump to subroutine starting at address represented by label
	CALL	cond.label	3							Jump to subroutine if condition is satisfied; otherwise, continue in sequence.
	RET		1							[PC(LO)] ← [[SP]] [PC(HI)] ← [[SP] + 1] [SP] ← [SP] + 2 Return from subroutine.
	RET	cond	1							Return from subroutine if condition is satisfied; otherwise, continue in sequence

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
OPERAZIONE IMMEDIATA	ADD	data	2	X	X	X	O	X	0	[A]←[A] + data Add immediate to Accumulator.
	ADC	data	2	X	X	X	O	X	0	[A]←[A] + data + C Add immediate with Carry.
	SUB	data	2	X	X	X	O	X	1	[A]←[A] - data Subtract immediate from Accumulator.
	SBC	data	2	X	X	X	O	X	1	[A]←[A] - data - C Subtract immediate with Carry.
	AND	data	2	0	X	X	P	1	0	[A]←[A] Λ data AND immediate with Accumulator.
	OR	data	2	0	X	X	P	1	0	[A]←[A] V data OR immediate with Accumulator.
	XOR	data	2	0	X	X	P	1	0	[A]←[A] ⊻ data Exclusive-OR immediate with Accumulator.
	CP	data	2	X	X	X	O	X	1	[A] - data Compare immediate data with Accumulator contents; only the flags are affected.
SALTO SU CONDIZIONE	JP	cond,label	3							If cond, then [PC]← label Jump to instruction at address represented by label if the condition is true.
	JR	C,disp	2							If C=1, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Carry flag is set.
	JR	NC,disp	2							If C=0, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Carry flag is reset.
	JR	Z,disp	2							If Z=1, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Zero flag is set.
	JR	NZ,disp	2							If Z=0, then [PC]←[PC] + 2 + disp Jump relative to contents of Program Counter if Zero flag is reset.
	DJNZ	disp	2							[B]←[B] - 1 If [B] ≠ 0, then [PC]←[PC] + 2 + disp Decrement contents of B and Jump relative to contents of Program Counter if result is not 0.

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO C	Z	S	P/O	A_C	N	OPERAZIONE EFFETTUATA
SPOSTAMENTO DA REGISTRO A REGISTRO	LD	dst,src	1							[dst]←[src] Move contents of source register to destination register. Register designations src and dst may each be A, B, C, D, E, H or L.
	LD	A,IV	2		X	X	I	0	0	[A]←[IV] Move contents of Interrupt Vector register to Accumulator.
	LD	A,R	2		X	X	I	0	0	[A]←[R] Move contents of Refresh register to Accumulator.
	LD	IV,A	2							[IV]←[A] Load Interrupt Vector register from Accumulator.
	LD	R,A	2							[R]←[A] Load Refresh register from Accumulator.
	LD	SP,HL	1							[SP]←[HL] Move contents of HL to Stack Pointer.
	LD	SP,xy	2							[SP]←[xy] Move contents of Index register to Stack Pointer.
	EX	DE,HL	1							[DE]←→[HL] Exchange contents of DE and HL.
	EX	AF,AF'	1							[AF]←→[AF'] Exchange program status and alternate program status.
	EXX		1							([BC] [DE] [HL]) ←→ ([BC'] [DE'] [HL']) Exchange register pairs and alternate register pairs.

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
OPERAZIONE TRA REGISTRO E REGISTRO	ADD	reg	1	X	X	X	O	X	0	[A]←[A] + [reg] Add contents of register to Accumulator.
	ADC	reg	1	X	X	X	O	X	0	[A]←[A] + [reg] + C Add contents of register and Carry to Accumulator.
	SUB	reg	1	X	X	X	O	X	1	[A]←[A] - [reg] Subtract contents of register from Accumulator.
	SBC	reg	1	X	X	X	O	X	1	[A]←[A] - [reg] - C Subtract contents of register and Carry from Accumulator.
	AND	reg	1	0	X	X	P	1	0	[A]←[A] Λ [reg] AND contents of register with contents of Accumulator.
	OR	reg	1	0	X	X	P	1	0	[A]←[A] V [reg] OR contents of register with contents of Accumulator.
	XOR	reg	1	0	X	X	P	1	0	[A]←[A] ⊻ [reg] Exclusive-OR contents of register with contents of Accumulator.
	CP	reg	1	X	X	X	O	X	1	[A] - [reg] Compare contents of register with contents of Accumulator. Only the flags are affected.
	ADD	HL,rp	1	X				?	0	[HL]←[HL] + [rp] 16-bit add register pair contents to contents of HL.
	ADC	HL,rp	2	X	X	X	O	?	0	[HL]←[HL] + [rp] + C 16-bit add with Carry register pair contents to contents of HL.
	SBC	HL,rp	2	X	X	X	O	?	1	[HL]←[HL] - [rp] - C 16-bit subtract with Carry register pair contents from contents of HL.
	ADD	IX,pp	2	X				?	0	[IX]←[IX] + [pp] 16-bit add register pair contents to contents of Index register IX (pp=BC, DE, IX, SP)
	ADD	IY,rr	2	X				?	0	[IY]←[IY] + [rr] 16-bit add register pair contents to contents of Index register IY (rr=BC, DE, IY, SP).

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDI (I)	BYTE	STATO C	STATO Z	STATO S	STATO P/O	STATO A_C	STATO N	OPERAZIONE EFFETTUATA
OPERAZIONE DI REGISTRI	DAA		1	X	X	X	P	X		Decimal adjust Accumulator, assuming that Accumulator contents are the sum or difference of BCD operands.
	CPL		1					1	1	[A]←[$\overline{A}$] Complement Accumulator (ones complement).
	NEG		2	X	X	X	O	X	1	[A]←[$\overline{A}$] + 1 Negate Accumulator (twos complement).
	INC	reg	1		X	X	O	X	0	[reg]←[reg] + 1 Increment register contents.
	INC	rp xy	1							[rp]←[rp] + 1 or [xy]←[xy] + 1 Increment contents of register pair or Index register.
	DEC	reg	2		X	X	O	X	1	[reg]←[reg] - 1 Decrement register contents.
	DEC	rp xy	1 2							[rp]←[rp] - 1 or [xy]←[xy] - 1 Decrement contents of register pair or Index register.
ROTAZIONE E SPOSTAMENTO DI REGISTRI	RLCA		1	X				0	0	C ← 7 ← 0 [A] Rotate Accumulator left with branch Carry.
	RLA		1	X				0	0	C ← 7 ← 0 [A] Rotate Accumulator left through Carry.
	RRCA		1	X				0	0	7 → 0 → C [A] Rotate Accumulator right with branch Carry.

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO C	STATO Z	STATO S	STATO P/O	STATO A_C	STATO N	OPERAZIONE EFFETTUATA
ROTAZIONE E SPOSTAMENTO DI REGISTRI (Continua)	RRA		1	X				0	0	[A] Rotate Accumulator right through Carry.
	RLC	reg	2	X	X	X	P	0	0	[reg] Rotate contents of register left with branch Carry
	RL	reg	2	X	X	X	P	0	0	[reg] Rotate contents of register left through Carry.
	RRC	reg	2	X	X	X	P	0	0	[reg] Rotate contents of register right with branch Carry.
	RR	reg	2	X	X	X	P	0	0	[reg] Rotate contents of register right through Carry.
	SLA	reg	2	X	X	X	P	0	0	[reg] Shift contents of register left and clear LSB (Arithmetic Shift)
	SRA	reg	2	X	X	X	P	0	0	[reg] Shift contents of register right and preserve MSB (Arithmetic Shift).

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICA	OPERANDO (I)	BYTE	STATO C	Z	S	P/O	A_C	N	OPERAZIONE EFFETTUATA
ROTAZIONE E SPOSTAMENTO DI REGISTRI (Continua)	SRL		2	X	X	X	P	0	0	0 → [7 → 0] → C [reg] Shift contents of register right and clear MSB (Logical Shift).
	RLD		2		X	X	P	0	0	[7 4|3 0] [7 4|3 0] [A] [[HL]] Rotate one BCD digit left between the Accumulator and memory location (implied addressing). Contents of the upper half of the Accumulator are not affected.
	RRD	reg	2		X	X	P	0	0	[7 4|3 0] [7 4|3 0] [A] [[HL]] Rotate one BCD digit right between the Accumulator and memory location (implied addressing). Contents of the upper half of the Accumulator are not affected.
MANIPOLAZIONE DI BIT	BIT	b,reg	2		X	?	?	1	0	$Z \leftarrow \overline{reg(b)}$ Zero flag contains complement of the selected register bit.
	BIT	b,(HL) b,(xy + disp)	2 4		X	?	?	1	0	$Z \leftarrow \overline{[[HL]](b)}$ or $Z \leftarrow \overline{[[xy]+disp](b)}$ Zero flag contains complement of selected bit of the memory location (implied addressing or base relative addressing).
	SET	b,reg	2							reg(b) ← 1 Set indicated register bit.
	SET	b,(HL) b,(xy + disp)	2 4							[[HL]](b) ← 1 or [[xy] + disp](b) ← 1 Set indicated bit of memory location (implied addressing or base relative addressing).
	RES	b,reg	2							reg(b) ← 0 Reset indicated register bit.
	RES	b,(HL) b,(xy + disp)	2 4							[[HL]](b) ← 0 or [[xy] + disp](b) ← 0 Reset indicated bit in memory location (implied addressing or base relative addressing).

Tabella 6-1. Sommario dell'insieme di istruzioni dello Z80 (segue)

TIPO	MNEMONICO	OPERANDO (I)	BYTE	STATO						OPERAZIONE EFFETTUATA
				C	Z	S	P/O	A_C	N	
STACK	PUSH	pr xy	1 2							[[SP]-1]←[pr(HI)] [[SP]-2]←[pr(LO)] [SP]←[SP]-2 Put contents of register pair or Index register on top of Stack and decrement Stack Pointer.
	POP	pr xy	1 2							[pr(LO)]←[[SP]] [pr(HI)]←[[SP]+1] [SP]←[SP]+2 Put contents of top of Stack in register pair or Index register and increment Stack Pointer.
	EX	(SP),HL (SP),xy	1 2							[H]←[[SP]+1] [L]←[[SP]] Exchange contents of HL or Index register and top of Stack.
INTERRUZIONE	DI		1							Disable interrupts.
	EI		1							Enable interrupts.
	RST	n	1							[[SP]-1]←[PC(HI)] [[SP]-2]←[PC(LO)] [SP]←[SP]-2 [PC]←$(8 \cdot n)_{16}$ Restart at designated location.
	RETI		2							Return from interrupt.
	RETN		2							Return from nonmaskable interrupt.
	IM	0 1 2	2							Set interrupt mode 0, 1, or 2.
STATO	SCF		1	1				0	0	C←1 Set Carry flag.
	CCF		1	X				?	0	C←$\overline{C}$ Complement Carry flag.
	NOP		1							No operation — volatile memories are refreshed.
	HALT		1							CPU halts, executes NOPs to refresh volatile memories.

Tabella 6-2 Sommario dei cicli di esecuzione e dei codici oggetto delle istruzioni

ISTRUZIONE		CODICE OGGETTO	BYTE	PERIODI DI CLOCK
ADC	data	CE yy	2	7
ADC	(HL)	8E	1	7
ADC	HL,rp	ED 01xx1010	2	15
ADC	(IX + disp)	DD 8E yy	3	19
ADC	(IY + disp)	FD 8E yy	3	19
ADC	reg	10001xxx	1	4
ADD	data	C6 yy	2	7
ADD	(HL)	86	1	7
ADD	HL,rp	00xx1001	1	11
ADD	(IX + disp)	DD 86 yy	3	19
ADD	IX pp	DD 00xx1001	2	15
ADD	(IY + disp)	FD 86 yy	3	19
ADD	IY,rr	FD 00xx1001	2	15
ADD	reg	10000xxx	1	4
AND	data	E6 yy	2	7
AND	(HL)	A6	1	7
AND	(IX + disp)	DD A6 yy	3	19
AND	(IY + disp)	FD A6 yy	3	19
AND	reg	10100xxx	1	4
BIT	b,(HL)	CB 01bbb110	2	12
BIT	b,(IX + disp)	DD CB yy 01bbb110	4	20
BIT	b,(IY + disp)	FD CB yy 01bbb110	4	20
BIT	b,reg	CB 01bbbxxx	2	8
CALL	label	CD ppqq	3	17
CALL	C,label	DC ppqq	3	10/17
CALL	M,label	FC ppqq	3	10/17
CALL	NC,label	D4 ppqq	3	10/17
CALL	NZ,label	C4 ppqq	3	10/17
CALL	P,label	F4 ppqq	3	10/17
CALL	PE,label	EC ppqq	3	10/17
CALL	PO,label	E4 ppqq	3	10/17
CALL	Z,label	CC ppqq	3	10/17
CCF		3F	1	4
CP	data	FE yy	2	7
CP	(HL)	BE	1	7
CP	(IX + disp)	DD BE yy	3	19
CP	(IY + disp)	FD BE yy	3	19
CP	reg	10111xxx	1	4
CPD		ED A9	2	16
CPDR		ED B9	2	21/16*
CPI		ED A1	2	16
CPIR		ED B1	2	21/16*
CPL		2F	1	4
DAA		27	1	4
DEC	(HL)	35	1	11
DEC	IX	DD 2B	2	10
DEC	(IX + disp)	DD 35 yy	3	23
DEC	IY	FD 2B	2	10
DEC	(IY + disp)	FD 35 yy	3	23
DEC	rp	00xx1011	1	6

Tabella 6-2. Sommario dei cicli di eseguzione e dei codici oggetto delle istruzioni (segue)

ISTRUZIONE		CODICE OGGETTO	BYTE	PERIODI DI CLOCK
DEC	reg	00xxx101	1	4
DI		F3	1	4
DJNZ	disp	10 yy	2	8/13
EI		FB	1	4
EX	AF,AF'	08	1	4
EX	DE,HL	EB	1	4
EX	(SP),HL	E3	1	19
EX	(SP),IX	DD E3	2	23
EX	(SP),IY	FD E3	2	23
EXX		D9	1	4
HALT		76	1	4
IM	0	ED 46	2	8
IM	1	ED 56	2	8
IM	2	ED 5E	2	8
IN	A,port	DB yy	2	10
IN	reg,(C)	ED 01ddd000	2	11
INC	(HL)	34	1	11
INC	IX	DD 23	2	10
INC	(IX + disp)	DD 34 yy	3	23
INC	IY	FD 23	2	10
INC	(IY + disp)	FD 34 yy	3	23
INC	rp	00xx0011	1	6
INC	reg	00xxx100	1	4
IND		ED AA	2	15
INDR		ED BA	2	20/15
INI		ED A2	2	15
INIR		ED B2	2	20/15
JP	label	C3 ppqq	3	10
JP	C,label	DA ppqq	3	10
JP	(HL)	E9	1	4
JP	(IX)	DD E9	2	8
JP	(IY)	FD E9	2	8
JP	M,label	FA ppqq	3	10
JP	NC,label	D2 ppqq	3	10
JP	NZ,label	C2 ppqq	3	10
JP	P,label	F2 ppqq	3	10
JP	PE,label	EA ppqq	3	10
JP	PO,label	E2 ppqq	3	10
JP	Z,label	CA ppqq	3	10
JR	C,disp	38 yy	2	7/12
JR	disp	18 yy	2	12
JR	NC,disp	30 yy	2	7/12
JR	NZ,disp	20 yy	2	7/12
JR	Z,disp	28 yy	2	7/12
LD	A,(addr)	3A ppqq	3	13
LD	A,(BC)	0A	1	7
LD	A,(DE)	1A	1	7
LD	A,I	ED 57	2	9
LD	A,R	ED 5F	2	9
LD	(addr),A	32 ppqq	3	13
LD	(addr),BC	ED 43 ppqq	4	20
LD	(addr),DE	ED 53 ppqq	4	20

Tabella 6-2. Sommario dei cicli di eseguzione e dei codici oggetto delle istruzioni (segue)

ISTRUZIONE		CODICE OGGETTO	BYTE	PERIODO DI CLOCK
LD	(addr),HL	22 ppqq	3	16
LD	(addr),IX	DD 22 ppqq	4	20
LD	(addr),IY	FD 22 ppqq	4	20
LD	(addr),SP	ED 73 ppqq	4	20
LD	(BC),A	02	1	7
LD	(DE),A	12	1	7
LD	HL,(addr)	2A ppqq	3	16
LD	(HL),data	36 yy	2	10
LD	(HL),reg	01110sss	1	7
LD	I,A	ED 47	2	9
LD	IX,(addr)	DD 2A ppqq	4	20
LD	IX,data16	DD 21 yyyy	4	14
LD	(IX + disp),data	DD 36 yy yy	4	19
LD	(IX + disp),reg	DD 01110sss yy	3	19
LD	IY,(addr)	FD 2A ppqq	4	20
LD	IY,data16	FD 21 yyyy	4	14
LD	(IY + disp),data	FD 36 yyyy	4	19
LD	(IY + disp),reg	FD 01110sss yy	3	19
LD	R,A	ED 4F	2	9
LD	reg,data	00ddd110 yy	2	7
LD	reg,(HL)	01ddd110	1	7
LD	reg,(IX + disp)	DD 01ddd110 yy	3	19
LD	reg,(IY + disp)	FD 01dddd110 yy	3	19
LD	reg,reg	01dddsss	1	4
LD	rp,(addr)	ED 01xx1011 ppqq	4	20
LD	rp,data16	00xx0001 yyyy	3	10
LD	SP,HL	F9	1	6
LD	SP,IX	DD F9	2	10
LD	SP,IY	FD F9	2	10
LDD		ED A8	2	16
LDDR		ED B8	2	21/16*
LDI		ED A0	2	16
LDIR		ED B0	2	21/16*
NEG		ED 44	2	8
NOP		00	1	4
OR	data	F6 yy	2	7
OR	(HL)	B6	1	7
OR	(IX + disp)	DD B6 yy	3	19
OR	(IY + disp)	FD B6 yy	3	19
OR	reg	10110xxx	1	4
OTDR		ED BB	2	20/15*
OTIR		ED B3	2	20/15*
OUT	(C),reg	ED 01sss001	2	12

Tabella 6-2. Sommario dei cicli di eseguzione e dei codici oggetto delle istruzioni (segue)

ISTRUZIONE	CODICE OGGETTO	BYTE	PERIODO DI CLOCK
OUT port,A	D3 yy	2	11
OUTD	ED AB	2	15
OUTI	ED A3	2	15
POP IX	DD E1	2	14
POP IY	FD E1	2	14
POP pr	11xx0001	1	10
PUSH IX	DD E5	2	15
PUSH IY	FD E5	2	15
PUSH pr	11xx0101	1	11
RES b,(HL)	CB 10bbb110	2	15
RES b,(IX + disp)	DD CB yy 10bbb110	4	23
RES b,(IY + disp)	FD CB yy 10bbb110	4	23
RES b,reg	CB 10bbbxxx	2	8
RET	C9	1	10
RET C	D8	1	5/11
RET M	F8	1	5/11
RET NC	D0	1	5/11
RET NZ	C0	1	5/11
RET P	F0	1	5/11
RET PE	E8	1	5/11
RET PO	E0	1	5/11
RET Z	C8	1	5/11
RETI	ED 4D	2	14
RETN	ED 45	2	14
RL (HL)	CB 16	2	15
RL (IX + disp)	DD CB yy 16	4	23
RL (IY + disp)	FD CB yy 16	4	23
RL reg	CB 00010xxx	2	8
RLA	17	1	4
RLC (HL)	CB 06	2	15
RLC (IX + disp)	DD CB yy 06	4	23
RLC (IY + disp)	FD CB yy 06	4	23
RLC reg	CB 00000xxx	2	8
RLCA	07	1	4
RLD	ED 6F	2	18
RR (HL)	CB 1E	2	15
RR (IX + disp)	DD CB yy 1E	4	23
RR (IY + disp)	FD CB yy 1E	4	23
RR reg	CB 00011xxx	2	8
RRA	1F	1	4
RRC (HL)	CB 0E	2	15
RRC (IX + disp)	DD CB yy 0E	4	23
RRC (IY + disp)	FD CB yy 0E	4	23

Tabella 6-2. Sommario dei cicli di esecuzione e dei codici oggetto delle istruzioni (segue)

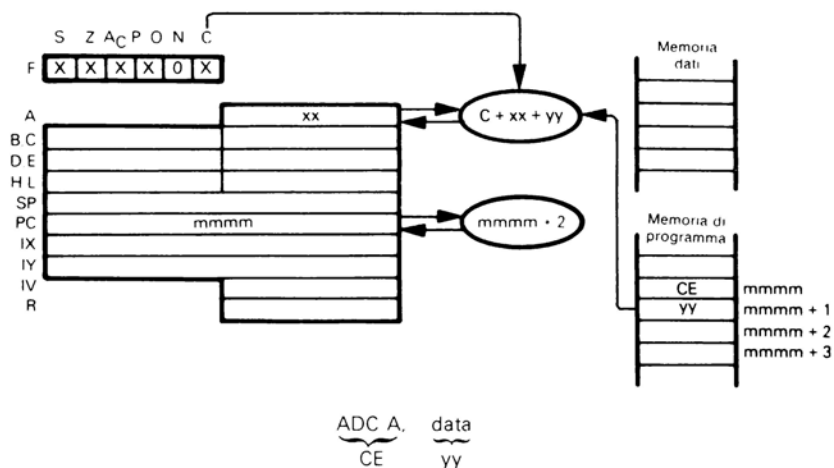
ISTRUZIONE		CODICE OGGETTO	BYTE	PERIODO DI CLOCK
RRC	reg	CB 00001xxx	2	8
RRCA		0F	1	4
RRD		ED 67	2	18
RST	n	11xxx111	1	11
SBC	data	DE yy	2	7
SBC	(HL)	9E	1	7
SBC	HL,rp	ED 01xx0010	2	15
SBC	(IX + disp)	DD 9E yy	3	19
SBC	(IY + disp)	FD 9E yy	3	19
SBC	reg	10011xxx	1	4
SCF		37	1	4
SET	b,(HL)	CB 11bbb110	2	15
SET	b,(IX + disp)	DD CB yy 11bbb110	4	23
SET	b,(IY + disp)	FD CB yy 11bbb110	4	23
SET	b,reg	CB 11bbbxxx	2	8
SLA	(HL)	CB 26	2	15
SLA	(IX + disp)	DD CB yy 26	4	23
SLA	(IY + disp)	FD CB yy 26	4	23
SLA	reg	CB 00100xxx	2	8
SRA	(HL)	CB 2E	2	15
SRA	(IX + disp)	DD CB yy 2E	4	23
SRA	(IY + disp)	FD CB yy 2E	4	23
SRA	reg	CB 00101xxx	2	8
SRL	(HL)	CB 3E	2	15
SRL	(IX + disp)	DD CB yy 3E	4	23
SRL	(IY + disp)	FD CB yy 3E	4	23
SRL	reg	CB 00111xxx	2	8
SUB	data	D6 yy	2	7
SUB	(HL)	96	1	7
SUB	(IX + disp)	DD 96 yy	3	19
SUB	(IY + disp)	FD 96 yy	3	19
SUB	reg	10010xxx	1	4
XOR	data	EE yy	2	7
XOR	(HL)	AE	1	7
XOR	(IX + disp)	DD AE yy	3	19
XOR	(IY + disp)	FD AE yy	3	19
XOR	reg	10101xxx	1	4

x rappresenta un digit binario opzionale.
bbb rappresenta digits binari opzionali identificanti l'allocazione di un bit in un byte di un registro o di memoria
ddd rappresenta digit binari opzionali identificanti un registro di destinazione
sss rappresenta digits binari opzionali identificanti un registro sorgente
ppqq rappresenta un indirizzo di memoria di quattro digit esadecimali
yy rappresenta due digits di un dato esadecimale
yyyy rappresenta quattro digits di un dato esadecimale

Quando si mostrano due possibili tempi di esecuzione (es. 5/11) ciò vuol dire che il numero dei periodi di clock dipende dai flags delle condizioni.

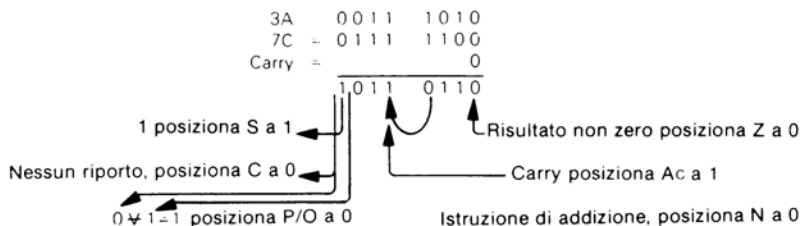
* Il tempo di esecuzione mostrato si riferisce ad una sola iterazione

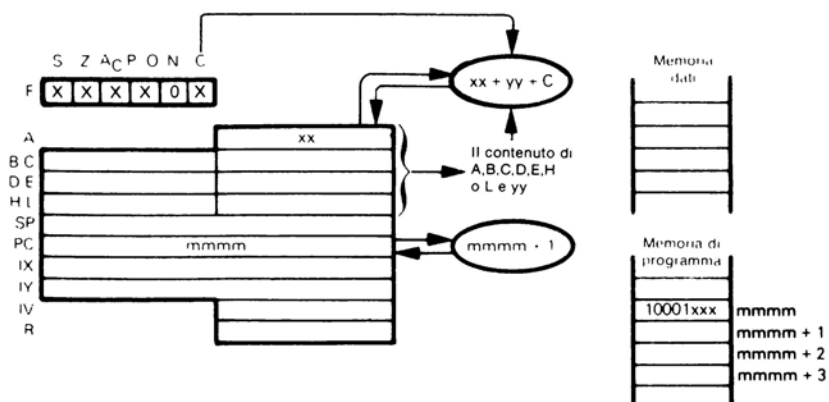
ADC, A,data – SOMMA IMMEDIATA CON CARRY ALL'ACCUMULATORE

ADC A, data
CE yy

Somma il contenuto del prossimo byte di memoria programmi e lo stato di Carry all'Accumulatore.

Supponiamo che xx=$3A_{16}$, yy=$7C_{16}$ e Carry=0. Dopo l'esecuzione dell'istruzione:

ADC A,7CH

l'Accumulatore conterrà $B6_{16}$:

L'istruzione ADC è frequentemente usata in una somma di più byte per il secondo e i byte seguenti.

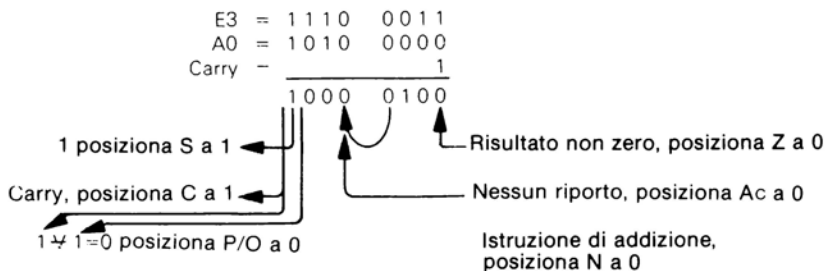
ADC A,reg — SOMMA IL REGISTRO CON CARRY ALL'ACCUMULATORE

ADC A, reg
10001 xxx

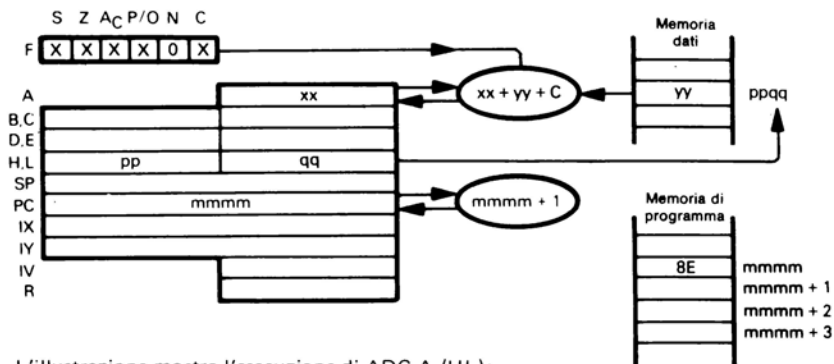
000 per reg = B
001 per reg = C
010 per reg = D
011 per reg = E
100 per reg = H
101 per reg = L
111 per reg = A

Somma il contenuto del Registro A, B, C, D, E, H o L e lo stato di Carry all'Accumulatore.

Supponiamo che xx=$E3_{16}$, che il Registro E contenga $A0_{16}$ e che il Carry=1. Dopo l'esecuzione dell'istruzione

ADC A,E

l'Accumulatore conterrà 84_{16}:

L'istruzione ADC è grandemente usata in una somma a più byte per il secondo ed i byte successivi.

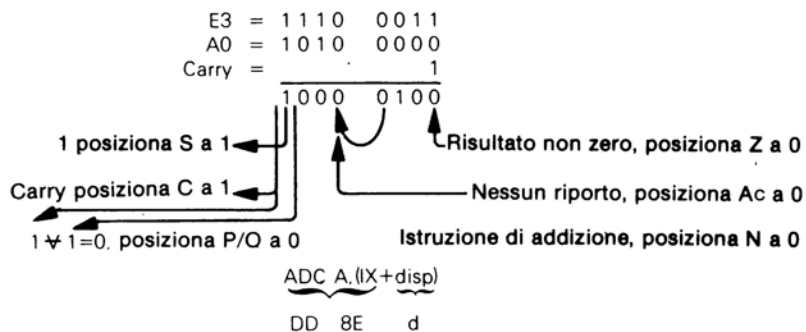
ADC A,(HL) — SOMMA DELLA MEMORIA E DEL CARRY
ADC A,(IX +disp)   ALL'ACCUMULATORE
ADC A,(IY +disp)

L'illustrazione mostra l'esecuzione di ADC A,(HL):

ADC A,(HL)
8E

Somma il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri H ed L) e lo stato del Carry all'Accumulatore.

Supponiamo che xx=$E3_{16}$, yy=$A0_{16}$ e Carry=1. Dopo l'esecuzione dell'istruzione

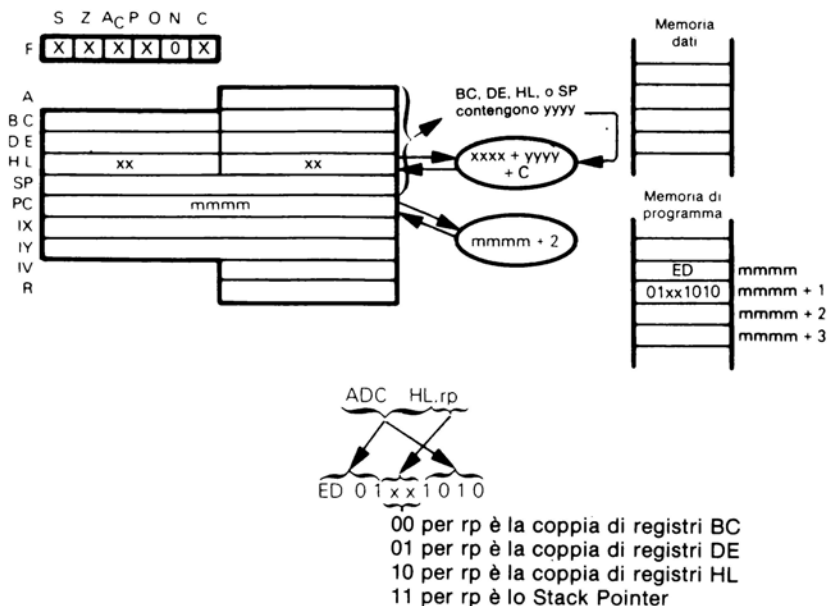
ADC A,(HL)

l'Accumulatore conterrà 84_{16}:

ADC A,(IX+disp)
DD 8E d

Somma il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del digit d di dislocamento) e del Carry all'Accumulatore.

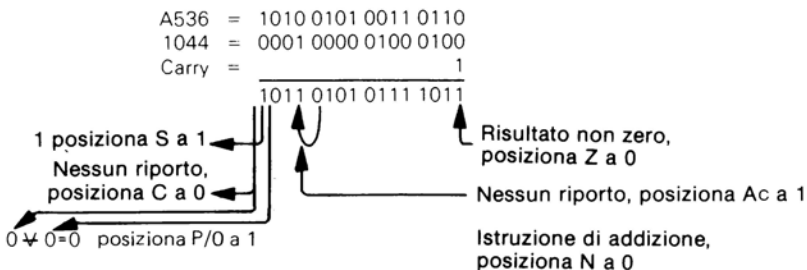
ADC A,(IY+disp)
FD 8E d

Questa istruzione è identica a ADC A,(IX+disp), tranne che essa usa il registro IY invece del registro IX. L'istruzione ADC è grandemente usata nelle somme con più byte per il secondo e i byte successivi.

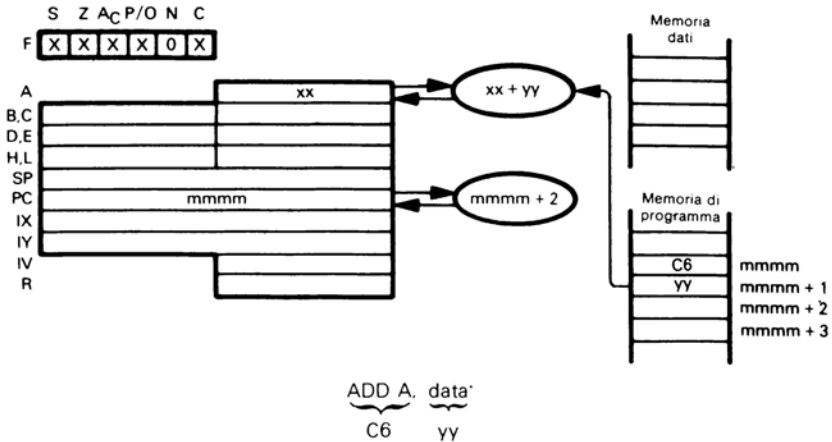
ADC HL,rp — SOMMA LA COPPIA DI REGISTRI CON CARRY AD H ED L

Somma il valore di 16 bit di una delle coppie di registri BC, DE, HL o dello Stack Pointer e lo stato di Carry alla coppia di registri H ed L.

Supponiamo che HL contenga $A536_{16}$, che BC contenga 1044_{16} e che Carry=1. Dopo l'esecuzione dell'istruzione

ADC HL,BC

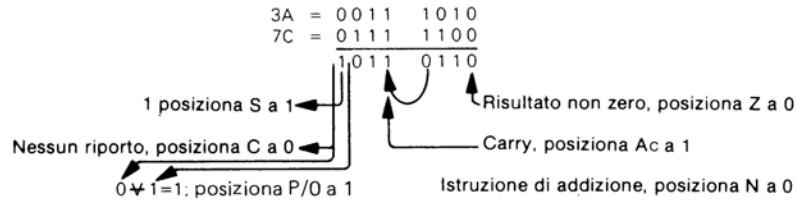
La coppia di registri H ed L conterrà:

L'istruzione ADC è grandemente usata in somme con più byte per il secondo e i byte successivi.

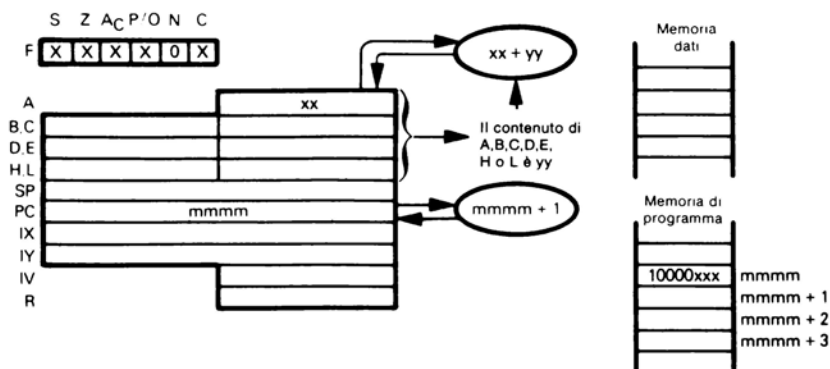
ADD A,data — SOMMA IMMEDIATA ALL'ACCUMULATORE

ADD A, data
C6 yy

Somma il contenuto del successivo byte della memoria di programma all'Accumulatore.

Supponiamo che xx=$3A_{16}$, yy=$7C_{16}$ e Carry=0. Dopo l'esecuzione dell'istruzione

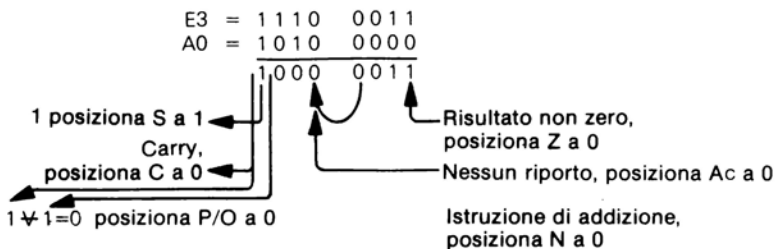
ADD A,7CH

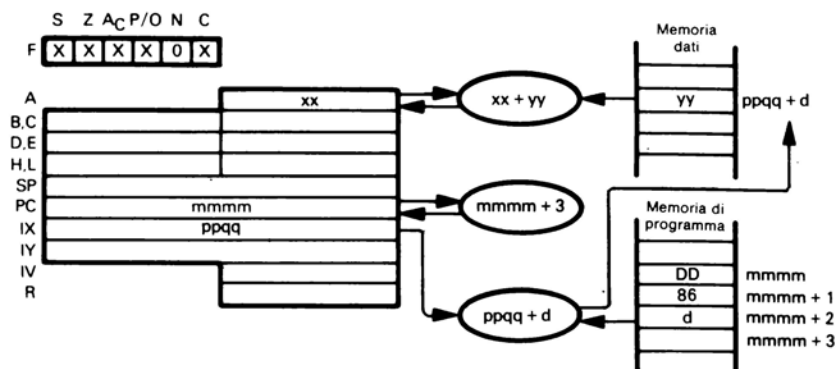
l'Accumulatore conterrà $B6_{16}$:

Questa è un'istruzione ordinaria di manipolazione dei dati.

ADD A,reg — SOMMA IL CONTENUTO DEL REGISTRO ALL'ACCUMULATORE

Somma il contenuto del Registro A, B, C, D, E, H o L all'Accumulatore.

Supponiamo che xx=$E3_{16}$. Il Registro E contenga $A0_{16}$. Dopo l'esecuzione di

ADD A,E

l'Accumulatore conterrà 83_{16}:

Questa è un'istruzione ordinaria di manipolazione dei dati.

ADD A,(HL) – SOMMA LA MEMORIA ALL'ACCUMULATORE
ADD A,(IX +disp)
ADD A,(IY +disp)

L'illustrazione mostra l'esecuzione di ADD A,(IX+disp).

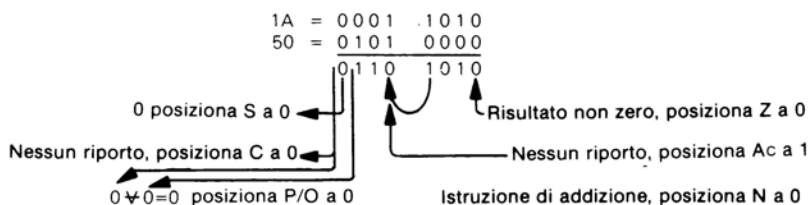
ADD A,(IX+disp)

DD 86 d

Somma il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del digit di dislocamento d) al contenuto dell'Accumulatore.

Supponiamo che ppqq=4000_{16}, xx=$1A_{16}$, e che la locazione di memoria $400F_{16}$ contenga 50_{16}. Dopo l'esecuzione dell'istruzione

ADD A,(IX+0FH)

l'Accumulatore conterrà $6A_{16}$.

ADD A,(IY+disp)

FD 86 d

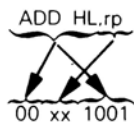
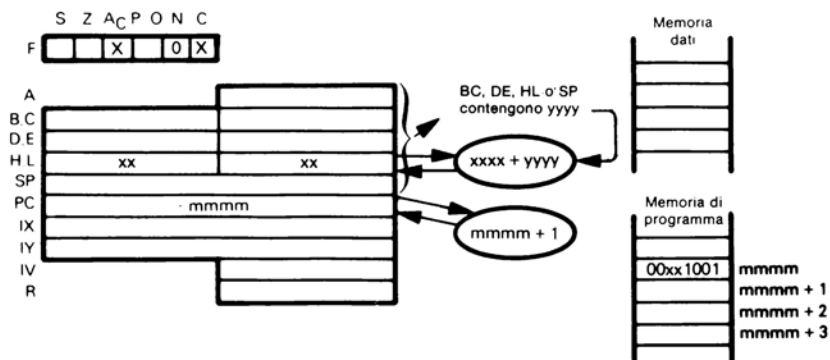
L'istruzione è identica a ADD A,(IX+disp), tranne che essa usa il registro IY invece del registro IX.

ADD A,(HL)

86

Questa versione dell'istruzione somma il contenuto della locazione di memoria, specificata dal contenuto della coppia di registri H ed L, all'Accumulatore.

L'istruzione ADD è un'istruzione ordinaria di manipolazione dei dati.

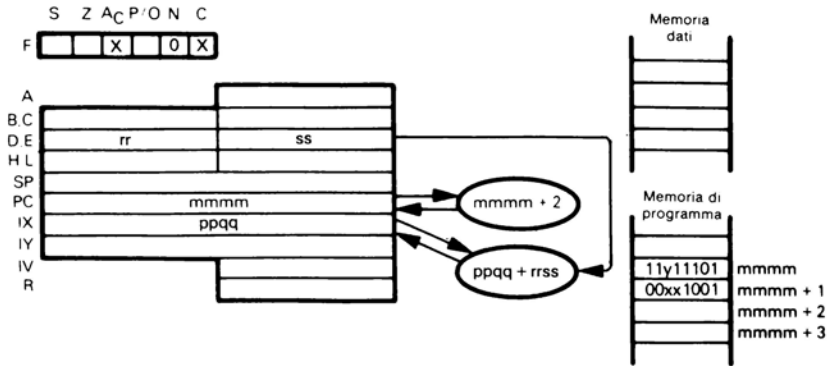
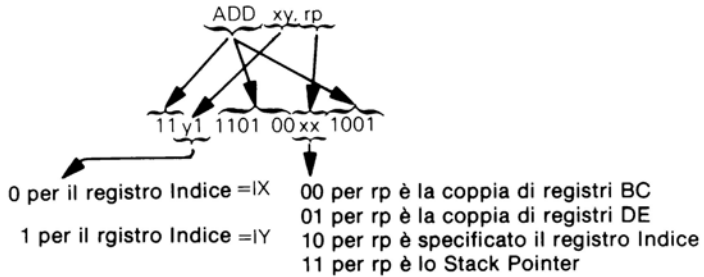
ADD HL,rp — SOMMA LA COPPIA DI REGISTRI AD H ED L

Somma il valore a 16 bit da una delle coppie di registri BC, DE, HL o dallo Stack Pointer alla coppia di registri H ed L.

Supponiamo che HL contenga $034A_{16}$ e che BC contenga $214C_{16}$. Dopo l'esecuzione dell'istruzione

ADD HL,BC

la coppia di registri HL conterrà 2496_{16}.

L'istruzione ADD HL,HL è equivalente a uno spostamento a sinistra dei 16 bit.

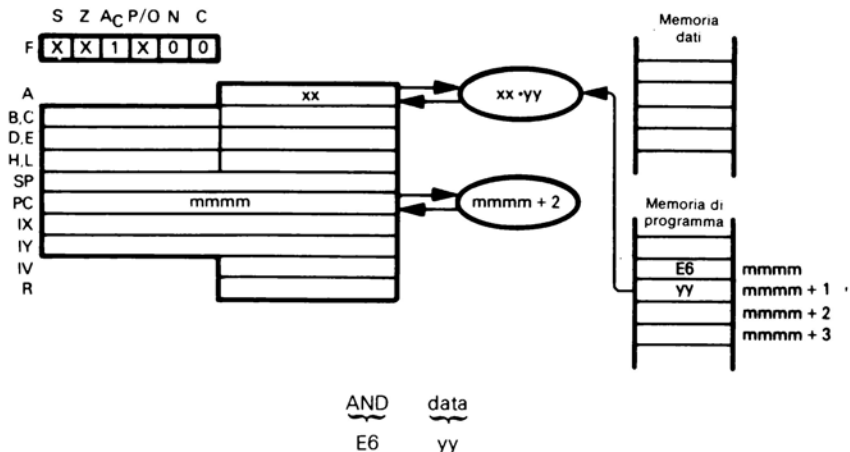
ADD xy,rp — SOMMA LA COPPIA DI REGISTRI AL REGISTRO INDICE

L'illustrazione mostra l'esecuzione di ADD IX,DE.

Somma il contenuto della coppia di registri specificata al contenuto del registro Indice specificato.

Supponiamo che IY contenga $4FF0_{16}$ e che BC contenga $000F_{16}$. Dopo l'esecuzione della istruzione

ADD IY,BC

il registro Indice IY conterrà $4FFF_{16}$.

AND data — AND IMMEDIATO CON L'ACCUMULATORE

AND data
E6 yy

AND del contenuto del successivo byte della memoria di programma con l'Accumulatore.

Supponiamo che xx=$3A_{16}$. Dopo l'esecuzione dell'istruzione

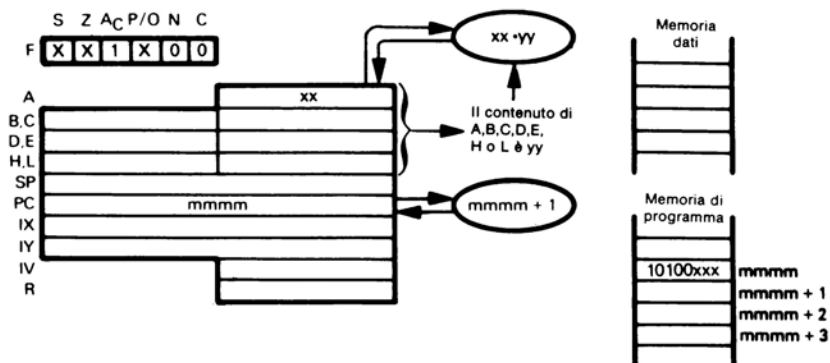
AND 7CH

l'Accumulatore conterrà 38_{16}.

Questa è un'istruzione logica ordinaria; è spesso usata per porre i bit nello stato "off". Per esempio, l'istruzione

AND 7FH

posizionerà incondizionatamente a 0 il bit di ordine maggiore dell'Accumulatore.

AND reg — AND DEL REGISTRO CON L'ACCUMULATORE

AND reg
10100 xxx

000 **per reg**=B
001 **per reg**=C
010 **per reg**=D
011 **per reg**=E
100 **per reg**=H
101 **per reg**=L
111 **per reg**=A

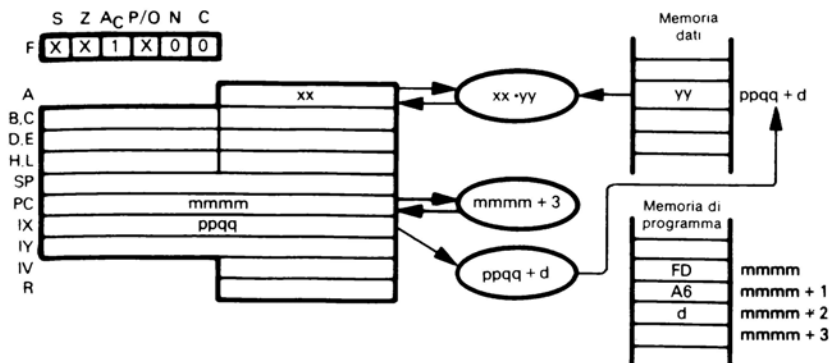
AND dell'Accumulatore col contenuto del Registro A, B, C, D, E, H o L. Salva il risultato nell'Accumulatore.

Supponiamo che xx=$E3_{16}$ e che il Registro E contenga $A0_{16}$. Dopo l'esecuzione della istruzione

AND E

l'Accumulatore conterrà $A0_{16}$.

AND è una istruzione logica usata frequentemente.

AND (HL) – AND DELLA MEMORIA CON L'ACCUMULATORE
AND (IX +disp)
AND (IY +disp)

L'illustrazione mostra l'esecuzione di AND (IY+disp).

AND (IY+disp)
FD A6 d

AND del contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IY e del digit d del dislocamento) con l'Accumulatore

Supponiamo che xx=$E3_{16}$, ppqq=4000_{16} e che la locazione di memoria $400F_{16}$ contenga $A0_{16}$. Dopo l'esecuzione dell'istruzione

AND (IY+0FH)

l'Accumulatore conterrà $A0_{16}$.

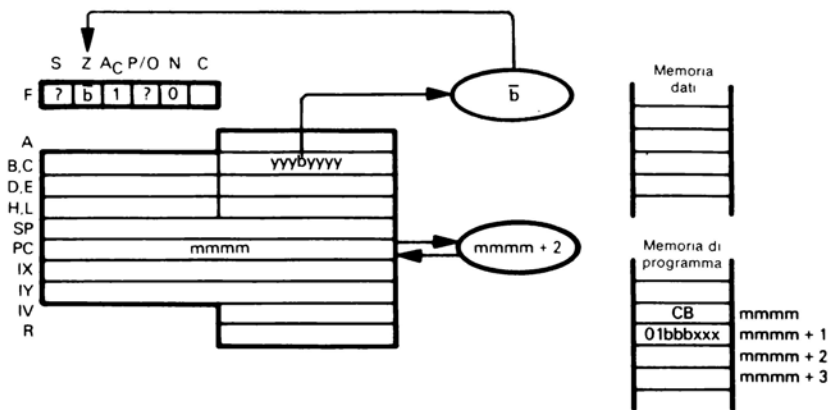
AND (IX+disp)
DD A6 d

Questa istruzione è identica a AND (IY+disp), tranne che essa usa il registro IX invece del registro IY.

AND (HL)
A6

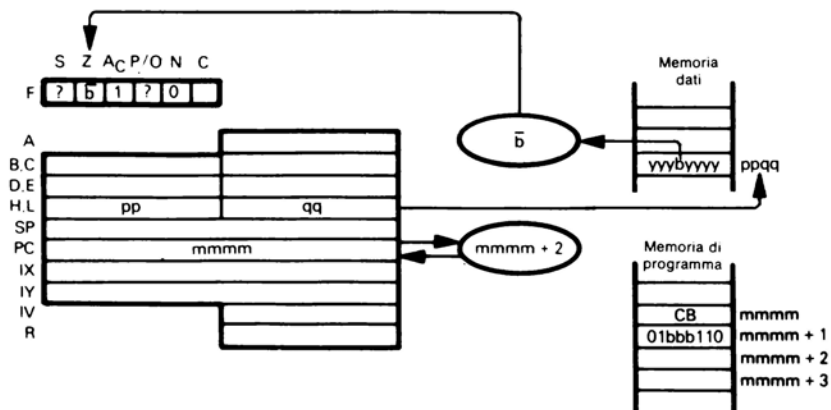
AND del contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) con l'Accumulatore.

AND è un'istruzione logica usata frequentemente.

BIT b,reg — TEST SUL BIT b NEL REGISTRO reg

Bit provato			Registro
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Pone il complemento del bit specificato del registro indicato nel flag Z del registro F.

Supponiamo che il Registro C contenga 1110 1111. L'istruzione BIT 4,C posizionerà allora ad 1 il flag Z, mentre il bit 4 nel Registro C rimane a 0. Il bit 0 è il bit meno significativo.

BIT b,(HL) — TEST SUL BIT b DELLA POSIZIONE DI MEMORIA INDICATA
BIT b,(IX+disp)
BIT b,(IY+disp)

L'illustrazione mostra l'esecuzione di BIT 4,(HL). Il bit 0 è il bit meno significativo.

BIT b (HL)

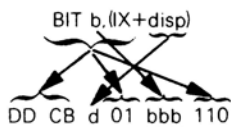
CB 01 bbb 110

Bit provato	bbb
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Prova il bit indicato nella posizione di memoria specificata dal contenuto del Registro HL e pone il complemento del bit nel flag Z del registro F.

Supponiamo che HL contenga 4000H e che il bit 3 nella locazione di memoria 4000H contenga 1. L'istruzione

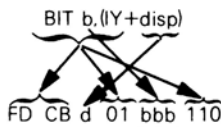
BIT 3,(HL)

posizionerà allora a 0 il flag Z, mentre il bit 3 nella locazione di memoria 4000H rimane ad 1.

bbb è lo stesso di BIT b, (HL)

Esamina il bit specificato nella locazione di memoria indicata dalla somma del Registro Indice IX e di disp. Pone il complemento nel flag Z del registro F.

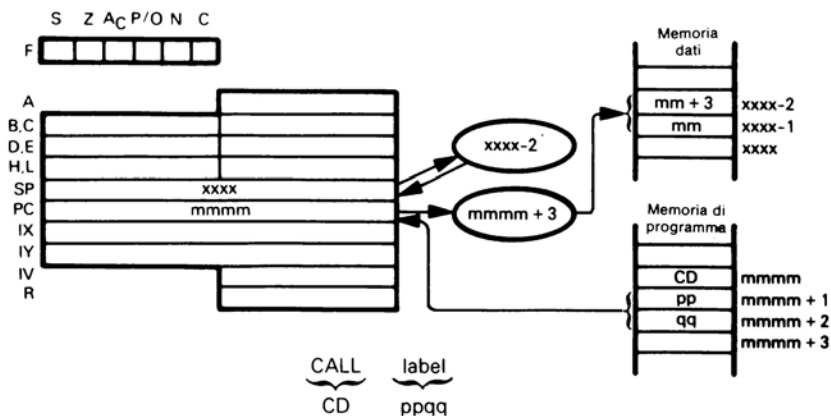
Supponiamo che il Registro IX contenga 4000H e che il bit 4 della locazione di memoria 4004H sia 0. L'istruzione

BIT 4,(IX + 4H)

posizionerà allora ad 1 il flag Z, mentre il bit 4 della locazione di memoria 4004H rimane a 0.

bbb è lo stesso di BIT b, (HL)

L'istruzione è identica a BIT b,(IX + disp), tranne che essa usa il registro IY invece del registro IX

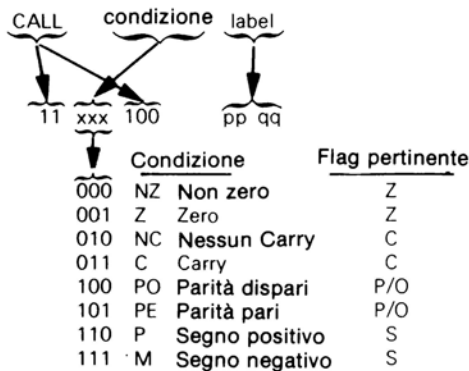
CALL label – CHIAMA IL SOTTOPROGRAMMA IDENTIFICATO NELL'OPERANDO

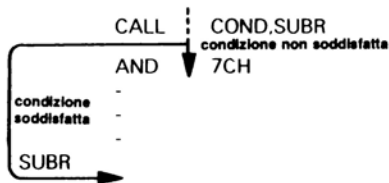
CALL label
CD ppqq

Memorizza l'indirizzo dell'istruzione che segue la CALL in cima allo stack: la cima dello stack è un byte di memoria dati indirizzato dallo Stack Pointer. Si sottrae poi 2 dallo Stack Pointer per indirizzare la nuova cima dello stack. Sposta l'indirizzo a 16 bit contenuto nei byte secondo e terzo del programma oggetto dell'istruzione CALL nel Contatore di Programma. Il secondo byte dell'istruzione CALL è la metà di ordine minore dell'indirizzo e il terzo byte è il byte di ordine maggiore.

Consideriamo la sequenza di istruzioni:

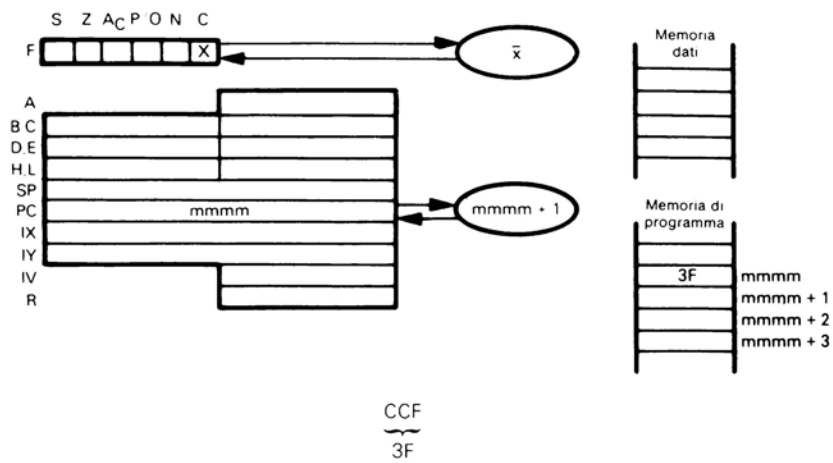
```
        CALL  SUBR
        AND   7CH
        –
        –
        –
SUBR
```

Dopo l'esecuzione dell'istruzione, l'indirizzo dell'istruzione AND è salvato in cima allo stack. Lo Stack Pointer è decrementato di 2. Successivamente si eseguirà l'istruzione SUBR.

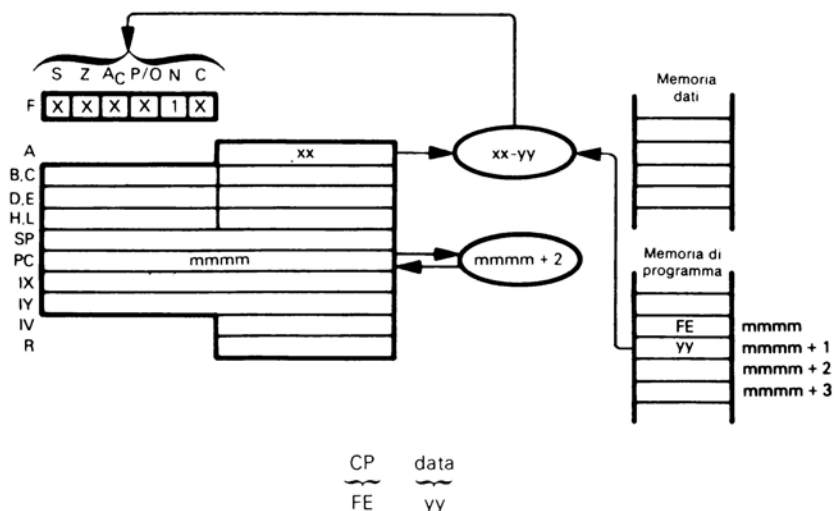
CALL condition, label — CHIAMA IL SOTTOPROGRAMMA IDENTIFICATO NELL'OPERANDO SE LA CONDIZIONE E' SODDISFATTA

Questa istruzione è identica all'istruzione CALL, tranne che il sottoprogramma identificato sarà chiamato solo se è soddisfatta la condizione; altrimenti sarà eseguita la istruzione che segue in sequenza l'istruzione CALL condition.

Consideriamo la sequenza di istruzioni:

Se la condizione non è soddisfatta, si eseguirà l'istruzione AND dopo l'esecuzione dell'istruzione CALL COND,SUBR. Se la condizione è soddisfatta, l'indirizzo della istruzione AND è salvato in cima allo stack e lo Stack Pointer è decrementato di 2. Successivamente si eseguirà l'istruzione avente l'etichetta SUBR.

CCF – COMPLEMENTA IL FLAG DI CARRY

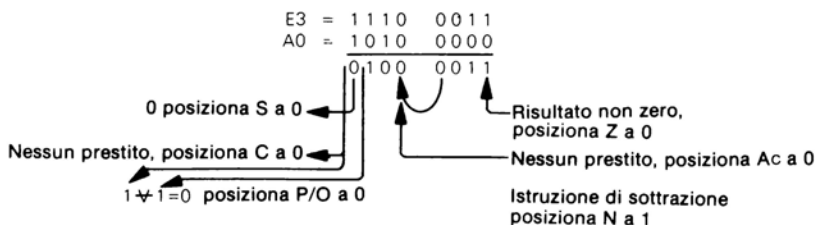
$$\underbrace{\text{CCF}}_{\text{3F}}$$

Complementa il flag di Carry. Non si influenza nessun altro stato o contenuto di registri.

CP data — CONFRONTA IMMEDIATAMENTE IL DATO COL CONTENUTO DELL'ACCUMULATORE

CP data
FE yy

Sottrae il contenuto del secondo byte del codice oggetto dal contenuto dell'Accumulatore, trattando entrambi i numeri come semplici dati binari. Scarta il risultato; cioè, lascia stare l'Accumulatore, ma modifica i flag degli stati per riflettere il risultato della sottrazione.

Supponiamo che xx=$E3_{16}$ e che il secondo byte del codice oggetto dell'istruzione CP contenga $A0_{16}$. Dopo l'esecuzione dell'istruzione

CP 0A0H

l'Accumulatore conterrà ancora $E3_{16}$, ma gli stati saranno modificati come segue:

E' da notare che il riporto risultante è complementato.

CP reg — CONFRONTA IL REGISTRO CON L'ACCUMULATORE

S Z Ac P/O N C
F X X X X 1 X

A xx
B,C
D,E
H,L
SP
PC mmmm
IX
IY
IV
R

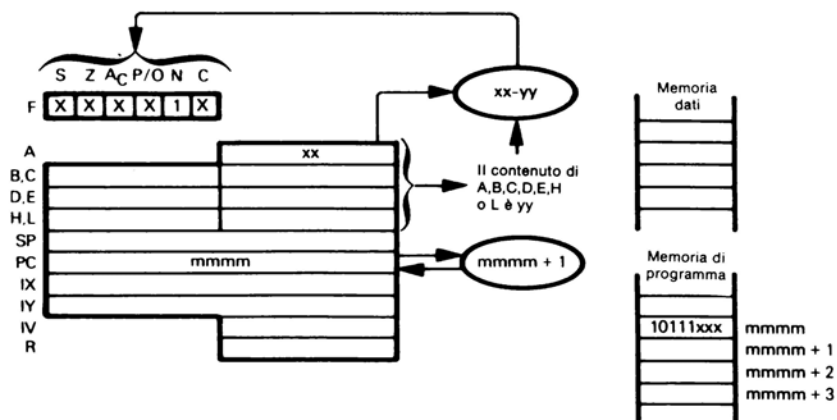
xx-yy

Il contenuto di A,B,C,D,E,H o L è yy

mmmm + 1

Memoria dati

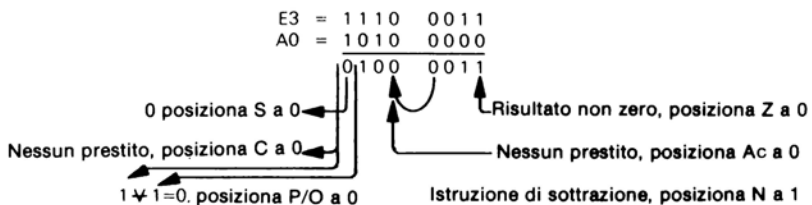
Memoria di programma

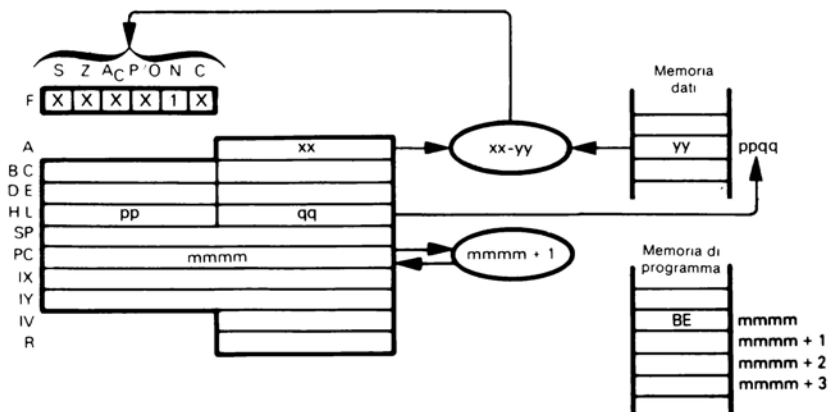
10111xxx mmmm
mmmm + 1
mmmm + 2
mmmm + 3

CP	reg
10111	xxx
	000 per reg = B
	001 per reg = C
	010 per reg = D
	011 per reg = E
	100 per reg = H
	101 per reg = L
	111 per reg = A

Sottrae il contenuto del Registro A, B, C, D, E, H o L dal contenuto dell'Accumulatore, trattando entrambi i numeri come semplici dati binari. Scarta il risultato; cioè lascia stare l'Accumulatore, ma modifica i flag degli stati per riflettere il risultato della sottrazione.

Supponiamo che xx=$E3_{16}$ e che il Registro B contenga $A0_{16}$. Dopo l'esecuzione della istruzione

CP B

l'Accumulatore conterrà ancora $E3_{16}$, ma gli stati saranno stati modificati come segue:

E' da notare che il carry risultante è complementato.

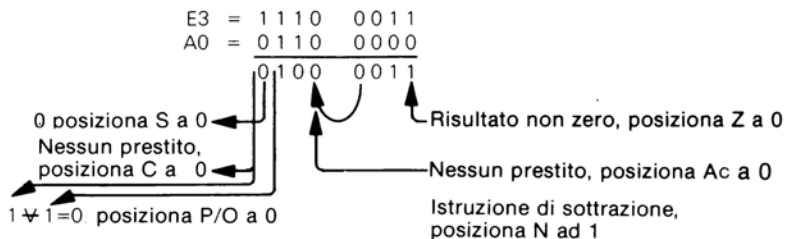
CP (HL) — CONFRONTA LA MEMORIA CON L'ACCUMULATORE
CP (IX +disp)
CP (IY +disp)

L'illustrazione mostra l'esecuzione di CP (HL):

CP (HL)
BE

Sottrae il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri H ed L) dal contenuto dell'Accumulatore, trattando entrambi i numeri come semplici numeri binari. Scarta il risultato; cioè, lascia stare l'Accumulatore, ma modifica i flag degli stati per riflettere il risultato della sottrazione.

Supponiamo che xx=$E3_{16}$ e yy=$A0_{16}$. Dopo l'esecuzione dell'istruzione

CP (HL)

l'Accumulatore conterrà ancora $E3_{16}$, ma gli stati saranno modificati come segue:

E' da notare che il carry risultante è complementato.

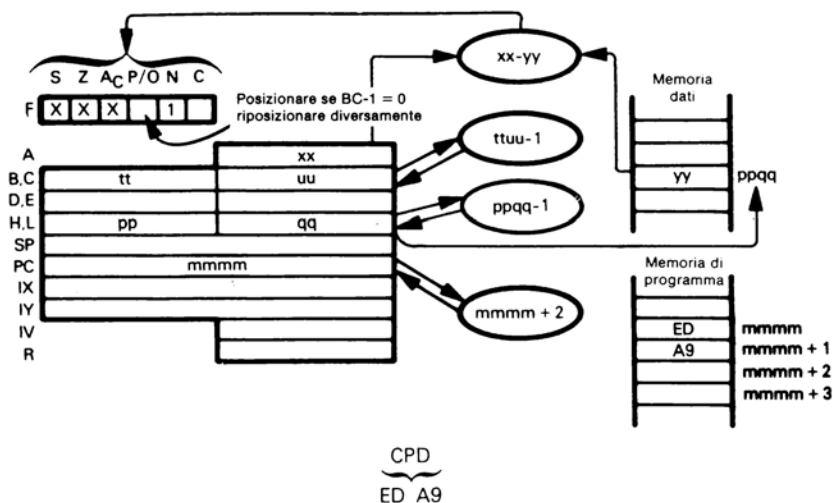
CP (IX+disp)
DD BE d

Sottrae il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX col valore d del dislocamento) dal contenuto dell'Accumulatore, trattando entrambi i numeri come semplici dati binari. Scarta il risultato; cioè, lascia stare l'Accumulatore, ma modifica i flag degli stati per riflettere il risultato della sottrazione.

CP (IY+disp)

FD BE d

Questa istruzione è identica a CP (IX + disp), tranne che essa usa il registro IY invece del registro IX.

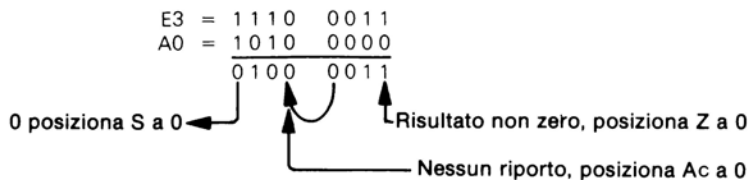
CPD — CONFRONTA L'ACCUMULATORE CON LA MEMORIA. DECREMENTA L'INDIRIZZO E IL CONTATORE DEI BYTE

CPD

ED A9

Confronta il contenuto dell'Accumulatore col contenuto della locazione di memoria (specificata dalla coppia di registri HL). Se A è uguale alla memoria, posiziona il flag Z. Decrementa la coppia di registri HL e BC. (BC è usato come Contatore dei Byte).

Supponiamo che xx=$E3_{16}$, ppqq=4000_{16}, che BC contenga 0001_{16} e che yy=$A0_{16}$. Dopo l'esecuzione dell'istruzione

CPD

l'Accumulatore conterrà ancora $E3_{16}$, ma gli stati saranno stati modificati come segue:

Il flag P/O sarà azzerato perché BC − 1 = 0

Istruzione di sottrazione interessata, posiziona N a 1

Il carry non è influenzato

La coppia di registri HL conterrà $3FFF_{16}$ e BC=0.

CPDR — CONFRONTA L'ACCUMULATORE CON LA MEMORIA. DECREMENTA L'INDIRIZZO E IL CONTATORE DEI BYTE. CONTINUA FINCHE' NON SI TROVA IL BYTE UGUALE O IL CONTATORE DEI BYTE E' ZERO

CPDR
ED B9

Questa istruzione è identica a CPD, tranne che essa è ripetuta finchè non si trovi un byte uguale o il contatore dei byte è a zero. Dopo il trasferimento di ogni dato, saranno riconosciute le interruzioni e saranno eseguiti due cicli di rinfresco (refresh).

Supponiamo che la coppia di registri HL contenga 5000_{16}, la coppia di registri BC contenga $00FF_{16}$, l'Accumulatore contenga $F9_{16}$ e che la memoria abbia il seguente contenuto:

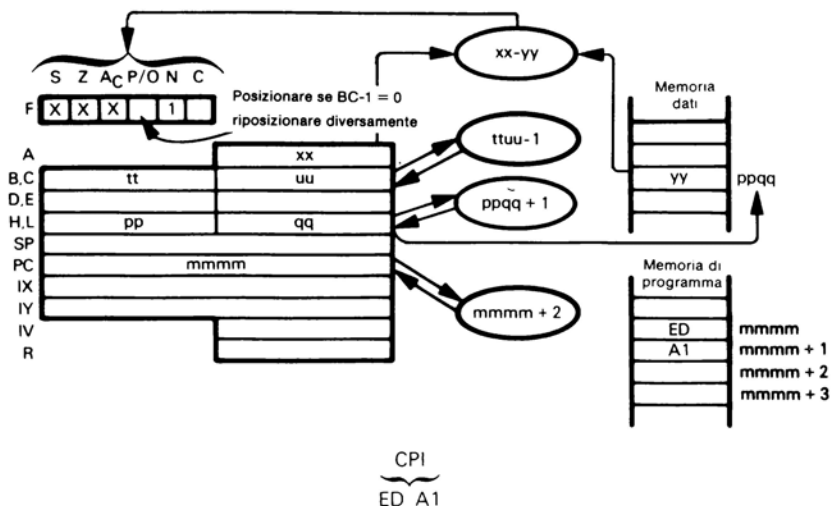
Locazione	Contenuto
5000_{16}	AA_{16}
$4FFF_{16}$	BC_{16}
$4FFE_{16}$	19_{16}
$4FFD_{16}$	$7A_{16}$
$4FFC_{16}$	$F9_{16}$
$4FFB_{16}$	DD_{16}

Dopo l'esecuzione dell'istruzione

CPDR

il flag P/O sarà 1, il flag Z sarà 1, la coppia di registri HL conterrà $4FFB_{16}$ e la coppia di registri BC conterrà $00FA_{16}$.

CPI — CONFRONTA L'ACCUMULATORE CON LA MEMORIA. DECREMENTA IL CONTATORE DEI BYTE. INCREMENTA L'INDIRIZZO

CPI
ED A1

Confronta il contenuto dell'Accumulatore col contenuto della locazione di memoria (specificata dalla coppia di registri HL). Se A è uguale alla memoria, posiziona ad 1 il flag Z. Incrementa la coppia di registri HL e decrementa la coppia di registri BC (BC è usata come Contatore dei Byte).

Supponiamo che xx=$E3_{16}$, ppqq=4000_{16}, che BC contenga 0032_{16} e che yy=$E3_{16}$. Dopo l'esecuzione dell'istruzione

CPI

l'Accumulatore conterrà ancora $E3_{16}$, ma gli stati saranno modificati come segue:

La coppia di registri HL conterrà 4001_{16} e BC conterrà 0031_{16}.

CPIR — CONFRONTA L'ACCUMULATORE CON LA MEMORIA. DECREMENTA IL CONTATORE DEI BYTE. INCREMENTA L'INDIRIZZO. CONTINUA FINCHE' NON SI TROVI UN BYTE UGUALE O IL CONTATORE DEI BYTE DIA ZERO

Questa istruzione è identica a CPI, tranne che essa è ripetuta finchè non si trovi un byte uguale o il contatore dei byte sia zero. Dopo ogni trasferimento di un dato, si riconosceranno le interruzioni e si eseguiranno due cicli di rinfresco.

Supponiamo che la coppia di registri HL contenga 4500_{16}, che la coppia di registri BC contenga $00FF_{16}$, e che l'Accumulatore contenga $F9_{16}$ e che la memoria abbia il seguente contenuto:

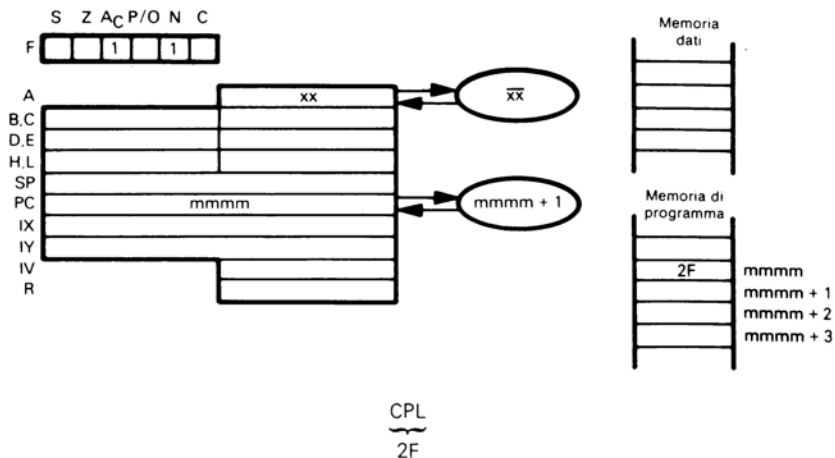
Locazione	Contenuto
4500_{16}	AA_{16}
4501_{16}	15_{16}
4502_{16}	$F9_{16}$

Dopo l'esecuzione di

CPIR

il flag P/O sarà 1 e il flag Z sarà 1. La coppia di registri HL conterrà 4503_{16} e la coppia di registri BC conterrà $00FC_{16}$.

CPL — COMPLEMENTA L'ACCUMULATORE

CPL
2F

Complementa il contenuto dell'Accumulatore. Non si influenza nessun altro contenuto di registri.

Supponiamo che l'Accumulatore contenga $3A_{16}$. Dopo l'esecuzione dell'istruzione

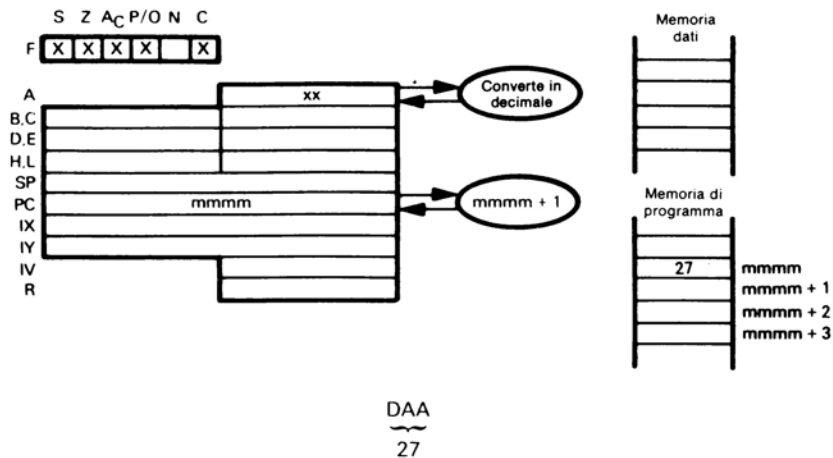
CPL

l'Accumulatore conterrà $C5_{16}$.

3A	=	0 0 1 1	1 0 1 0
Complemento	=	1 1 0 0	0 1 0 1

Questa è una istruzione logica ordinaria. Non bisogna usarla per sottrazione binaria; ci sono speciali istruzioni di sottrazione (SUB, SBC).

DAA — ADATTAMENTO DECIMALE DELL'ACCUMULATORE

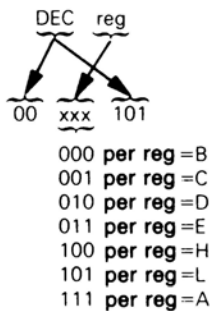
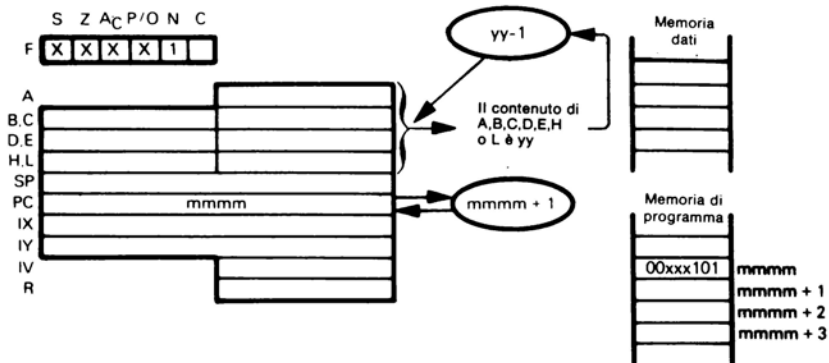
DAA
27

Converte il contenuto dell'Accumulatore in una forma binaria con codifica decimale. Questa istruzione potrebbe essere usata solo dopo la somma o la sottrazione di due numeri BCD; cioè, guardate ADD DAA o ADC DAA o INC DAA o SUB DAA o SBC DAA o DEC DAA o NEG DAA come istruzioni aritmetiche decimale composte, che operano su sorgenti BCD per generare risposte BCD.

Supponiamo che l'Accumulatore contenga 39_{16} e che il registro B contenga 47_{16}. Dopo l'esecuzione delle istruzioni

```
ADD B
DAA
```

l'Accumulatore conterrà 86_{16}, e non 80_{16}.

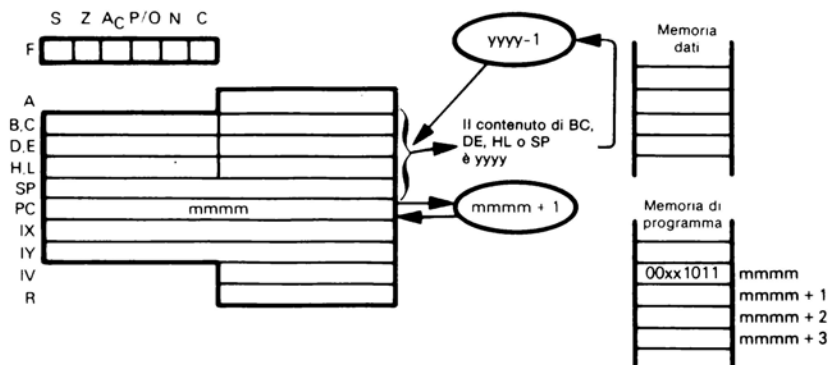
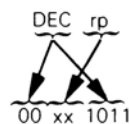
La logica della CPU Z80 usa i valori nel Carry e nel Ausiliary Carry, come pure il contenuto dell'Accumulatore, nell'operazione di Adattamento Decimale (Decimal Adjust).

DEC reg — DECREMENTA IL CONTENUTO DEL REGISTRO

Sottrae 1 dal contenuto del registro specificato.

Supponiamo che il Registro A contenga 50_{16}. Dopo l'esecuzione di

DEC A

il Registro A conterrà $4F_{16}$.

DEC rp — DECREMENTA IL CONTENUTO DELLA COPPIA
DEC IX   DI REGISTRI SPECIFICATA
DEC IY

L'illustrazione mostra l'esecuzione di DEC rp:

00 **per rp è la coppia di registri BC**
01 **per rp è la coppia di registri DE**
10 **per rp è la coppia di registri HL**
11 **per rp è lo Stack Pointer**

Sottrae 1 dal valore a 16 bit contenuto nella coppia di registri specificata. Non si influenza nessun flag di stato.

Supponiamo che i registri H ed L contengano $2F00_{16}$; dopo l'esecuzione dell'istruzione

DEC HL

i registri H ed L conterranno $2EFF_{16}$.

DEC IY

FD 2B

Sottrae 1 dal valore a 16 bit contenuto nel registro IX.

DEC IX

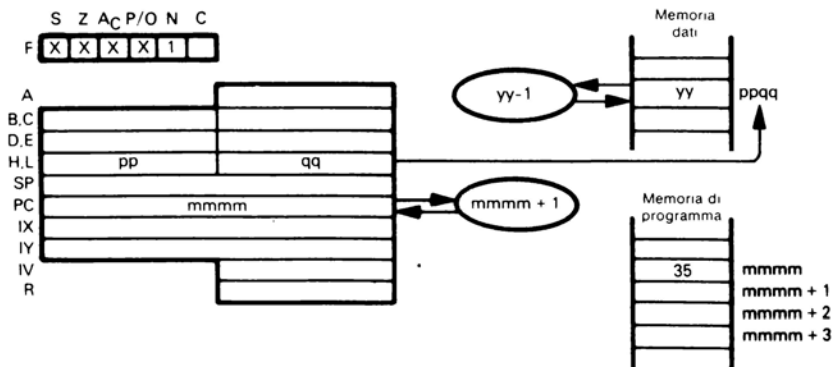
DD 2B

Sottrae 1 dal valore a 16 bit contenuto nel registro IY.

Nè DEC rp, nè DEC IX, nè DEC IY influenzano i flag di stato. Questo è un difetto dell'insieme delle istruzioni dello Z80, ereditato dallo 8080. Mentre si usa l'istruzione DEC reg in loop di istruzioni iterative che usano un contatore con valore 256 o minore, l'istruzione DEC rp (DEC IX o DEX IY) deve essere usata se il valore del contatore è maggiore di 256. Poichè l'istruzione DEC rp non posiziona nessun flag, si

devono aggiungere altre istruzioni semplicemente per verificare se il risultato è zero. Ecco una forma tipica di loop:

```
        LD    DE,DATA    ; Carica il valore iniziale del contatore a 16 bit
LOOP    –                ; Prima istruzione del LOOP
        –
        –
        DEC   DE         ; Decrementa il contatore
        LD    A,D        ; Per verificare se è zero, sposta D in A
        OR    E          ; Quindi fa l'OR di A con E
        JP    NZ,LOOP    ; Ritorna se non è zero
```

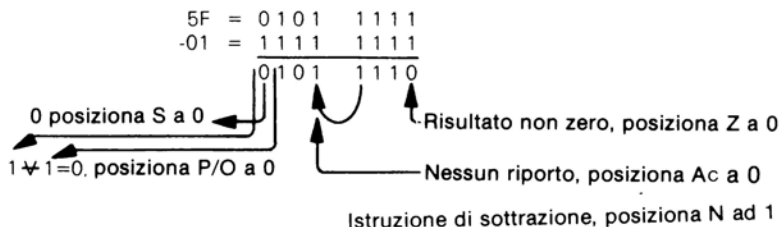
DEC (HL) – DECREMENTA IL CONTENUTO DELLA MEMORIA
DEC (IX +disp)
DEC (IY +disp)

L'illustrazione mostra l'esecuzione di DEC (HL):

$$\underbrace{\text{DEC (HL)}}_{35}$$

Sottrae 1 dal contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL).

Supponiamo che ppqq=4500_{16}, yy=$5F_{16}$. Dopo l'esecuzione di

DEC (HL)

la locazione di memoria 4500_{16} conterrà $5E_{16}$.

DEC (IX+disp)

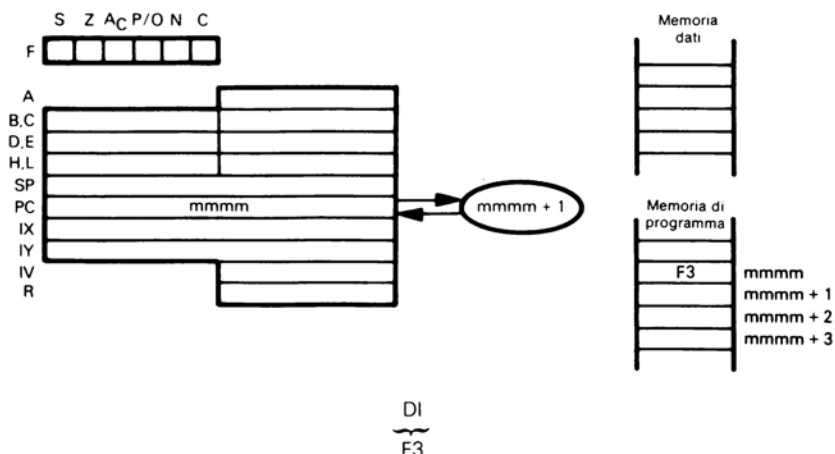
DD 35 d

Sottrae 1 dal contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX col valore d del dislocamento).

DEC (IY+disp)

FD 35 d

Questa istruzione è identica a DEC (IX + disp), tranne che essa usa il registro IY invece del reigstro IX.

DI – DISABILITA LE INTERRUZIONI

DI

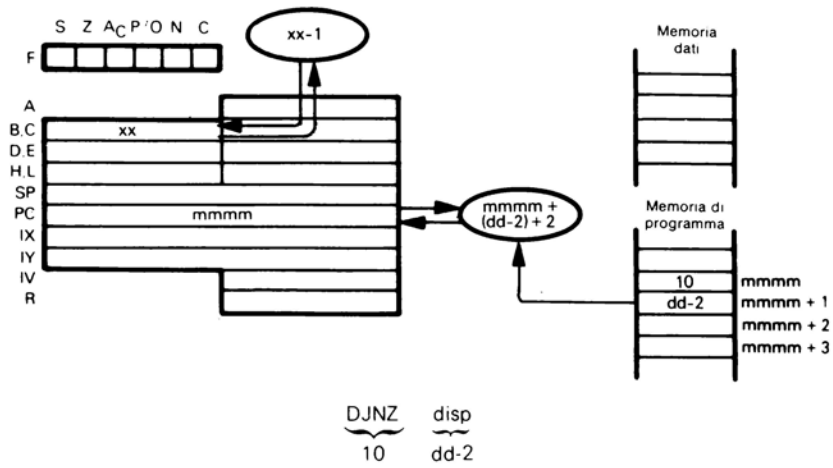
F3

Quando si esegue questa istruzione, si disabilita la richiesta delle interruzioni mascherabili e si ignorerà l'ingresso $\overline{INT}$ sulla CPU. E' da ricordare che quando un'interruzione viene riconosciuta, si disabilita automaticamente l'interruzione mascherabile.

La richiesta dell'interruzione mascherabile rimane disabilitata finchè essa non venga successivamente abilitata da un'istruzione EI.

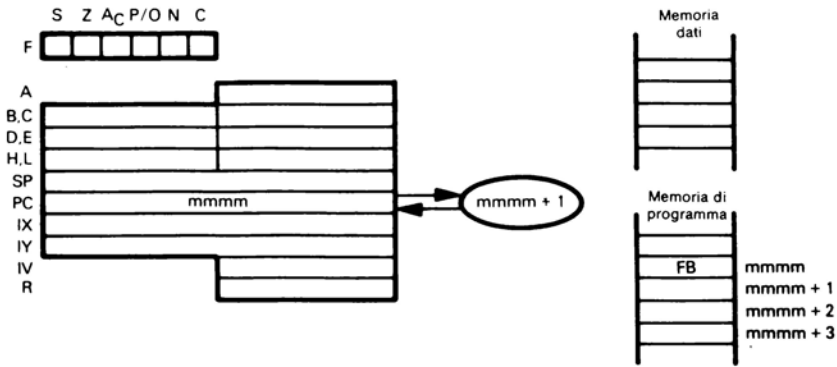
Nessun registro o flag è influenzato da questa istruzione.

DJNZ disp — SALTO RELATIVO AL CONTENUTO PRESENTE DEL CONTATORE DEI PROGRAMMI SE IL REG B NON E' ZERO

DJNZ disp
10 dd-2

Decrementa il Registro B. Se il contenuto che rimane non è zero, somma il contenuto del secondo byte del codice oggetto dell'istruzione DJNZ più 2 al Contatore di Programma. Il salto è misurato dall'indirizzo del codice operativo dell'istruzione, ed ha un campo compreso tra -126 e +129 byte. L'Assembler automaticamente si regola per l'incremento doppio del PC.

Se il contenuto di B è zero dopo il decremento, si esegue la successiva istruzione in sequenza.

L'istruzione DJNZ è estremamente utile per ogni operazione in un loop di un programma, poichè una sola istruzione sostituisce la tipica sequenza d'istruzioni "decrementa quindi salta su condizione".

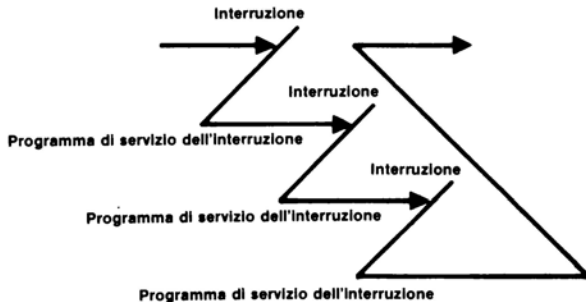
EI — ABILITA LE INTERRUZIONI

EI
FB

L'esecuzione di questa istruzione provoca la disabilitazione delle interruzioni, ma non finchè non si esegue almeno un'istruzione.

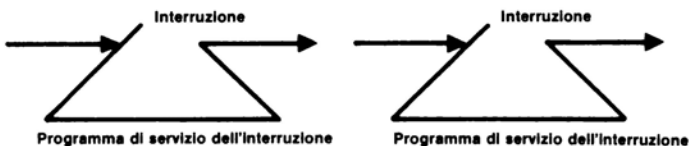
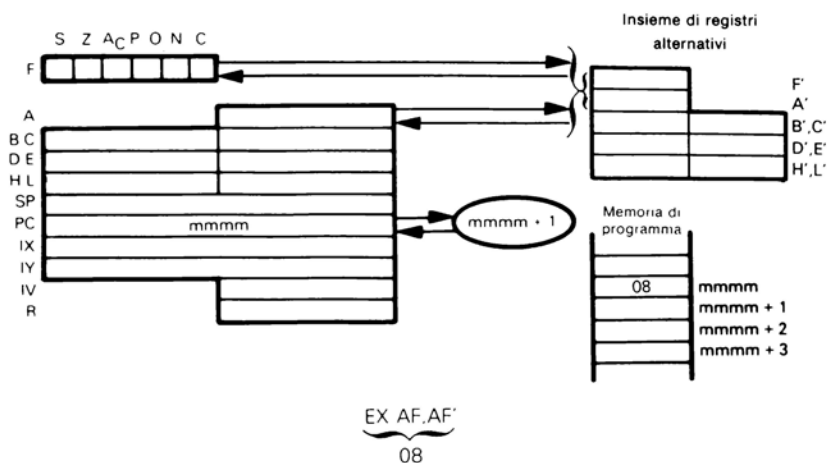
La maggior parte dei programmi di servizio delle interruzioni finisce con le due istruzioni:

```
        EI                  ; Abilita le interruzioni
        RET                 ; Ritorna al programma interrotto
```

Se le interruzioni subiscono un processo in serie, allora tutte le interruzioni mascherabili sono disabilitate per l'intera durata del programma di servizio delle interruzioni — che significa che in un'applicazione a multi-interruzione c'è una significativa possibilità che siano sospese una o più interruzioni quando un programma di servizio di un'interruzione completa la sua esecuzione.

Se le interruzioni fossero riconosciute non appena si è eseguita l'istruzione EI, allora non si eseguirebbe l'istruzione Return. In questo caso, i ritorni si accumulerebbero uno sopra l'altro — e necessariamente consumerebbero spazio della memoria della catasta (stack). Ciò può essere illustrato come segue:

Inibendo le interruzioni per almeno un'istruzione seguente l'esecuzione di EI, la CPU Z80 assicura che l'istruzione RET sarà eseguita in sequenza:

```
        —
        —
        —
        EI                  ; Abilita le interruzioni
        RET                 ; Ritorno dall'interruzione
```

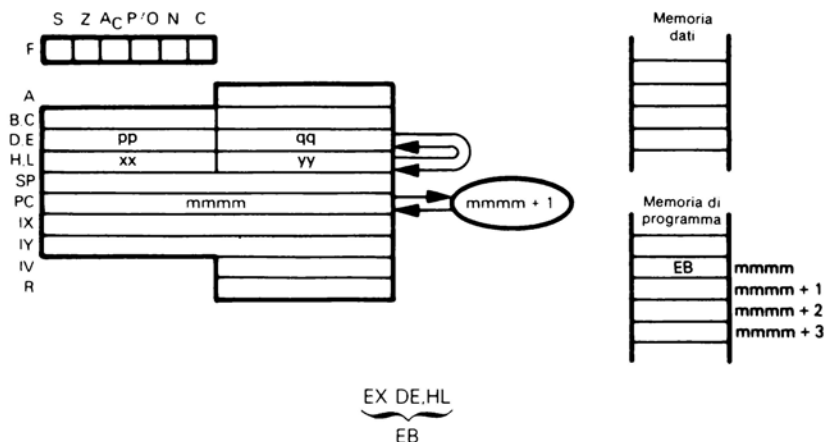
Non è raro che le interruzioni siano tenute disabilitate durante l'esecuzione di un programma di servizio di un'interruzione. Le interruzioni subiscono un processo serialmente:

EX AF,AF' – SCAMBIA LO STATO DEL PROGRAMMA E LO STATO DEL PROGRAMMA ALTERNATIVO

EX AF,AF'
08

Il contenuto dei due byte delle coppie di registri AF e A'F' vengono scambiati.

Supponiamo che AF contenga $4F99_{16}$ e che A'F' contenga $10AA_{16}$. Dopo l'esecuzione di

EX AF,AF'

AF conterrà $10AA_{16}$ e AF' conterrà $4F99_{16}$.

EX DE,HL — SCAMBIA I CONTENUTI DI DE ED HL

EX DE,HL
EB

Il contenuto dei registri D ed E è scambiato col contenuto dei registri H ed L.

Supponiamo che pp=03_{16}, qq=$2A_{16}$, xx=41_{16} e che yy=FC_{16}. Dopo l'esecuzione dell'istruzione

```
EX  DE,HL
```

H conterrà 03_{16}, L conterrà $2A_{16}$, D conterrà 41_{16} ed E conterrà FC_{16}.

Le due istruzioni:

```
EX  DE,HL
LD  A,(HL)
```

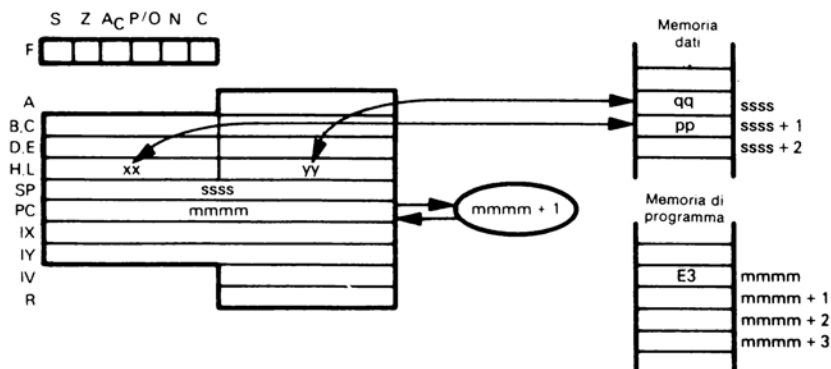
sono equivalenti a:

```
LD  A,(DE)
```

ma se volete caricare nel registro B il dato indirizzato dai registri D ed E,

```
EX  DE,HL
LD  B,(HL)
```

non hanno nessuna singola istruzione equivalente.

EX (SP),HL — SCAMBIA IL CONTENUTO DEL REGISTRO
EX (SP),IX   E DELLA SOMMITA' DELLO STACK
EX (SP),IY

L'illustrazione mostra l'esecuzione di EX (SP),HL.

$$\underbrace{\text{EX (SP),HL}}_{\text{E3}}$$

Scambia il contenuto del registro L con il byte in cima allo stack. Scambia il contenuto del registro H col byte che sta sotto alla cima dello stack.

Supponiamo che xx=21_{16}, yy=FA_{16}, pp=$3A_{16}$, qq=$E2_{16}$. Dopo l'esecuzione della istruzione

EX (SP),HL

H conterrà $3A_{16}$, L conterrà $E2_{16}$ e i due byte sulla sommità dello stack conterranno rispettivamente FA_{16} e 21_{16}.

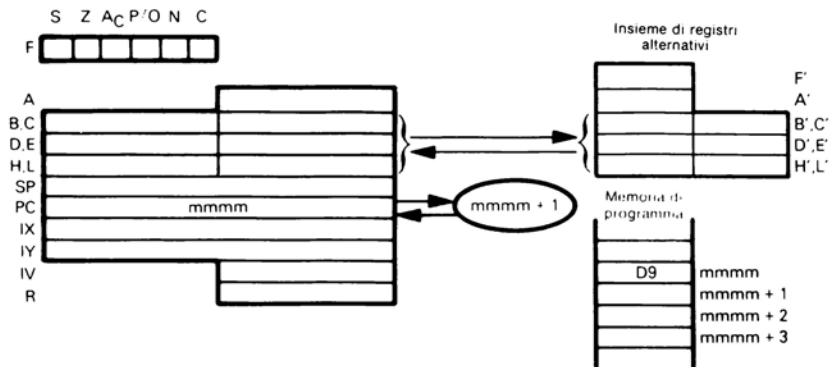
L'istruzione EX (SP),HL è usata per accedere e manipolare i dati sulla sommità dello stack.

$$\underbrace{\text{EX (SP),IX}}_{\text{DD E3}}$$

Scambia il contenuto del byte di ordine minore del registro IX col byte in cima allo stack. Scambia il byte di ordine maggiore del registro IX col byte che sta sotto alla sommità dello stack.

$$\underbrace{\text{EX (SP),IY}}_{\text{FD E3}}$$

Questa istruzione è identica a EX (SP),IX, ma usa il registro IY invece di IX.

EXX — SCAMBIA LA COPPIA DI REGISTRI CON LA COPPIA DI REGISTRI ALTERNATIVA

$$\underbrace{\text{EXX}}_{\text{D9}}$$

Il contenuto delle coppie di registri BC, DE e HL è scambiato col contenuto delle coppie di registri B'C', D'E' e H'L'.

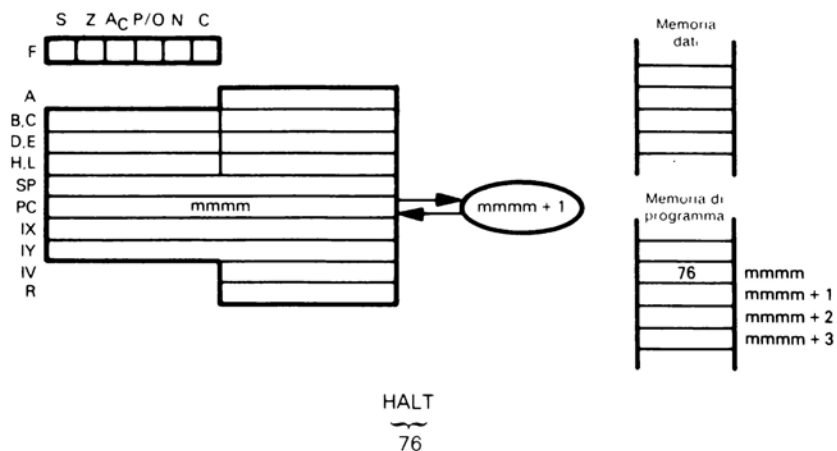
Supponiamo che le coppie di registri BC, DE e HL contengano rispettivamente 4901_{16}, $5F00_{16}$ e 7251_{16} e che le coppie di registri B'C', D'E' e H'L' contengano rispettivamente 0000_{16}, $10FF_{16}$ e 3333_{16}. Dopo l'esecuzione di

EXX

i registri avranno i seguenti contenuti:

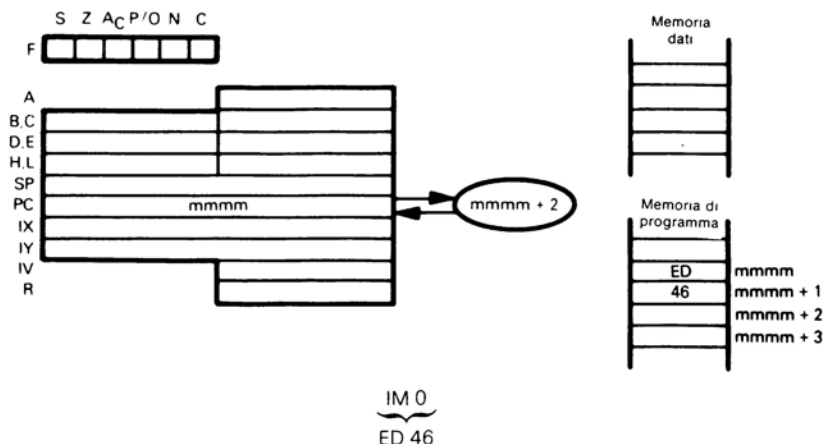
BC: 0000_{16}; DE: $10FF_{16}$; HL: 3333_{16};
B'C': 4901_{16}; D'E'. $5F00_{16}$; H'L': 7251_{16}

Questa istruzione può essere usata per scambiare banchi di registri per ottenere tempi di risposta alle interruzioni molto veloci.

HALT

HALT
76

Quando si esegue l'istruzione HALT, l'esecuzione del programma cessa. La CPU richiede un'interruzione o un reset per far ripartire l'esecuzione. Non vengono influenzati nè i registri nè gli stati; tuttavia, la logica di rinfresco della memoria continua a funzionare.

IM 0 — INTERRUZIONE DI MODO 0

IM 0
ED 46

Questa istruzione pone la CPU nel modo 0 d'interruzione. In questo modo, il dispositivo interrompente metterà un'istruzione sul Bus dei Dati e la CPU eseguirà quella istruzione. Non si influenza nessun registro o stato.

IM 1 — INTERRUZIONE DI MODO 1

IM 1
ED 56

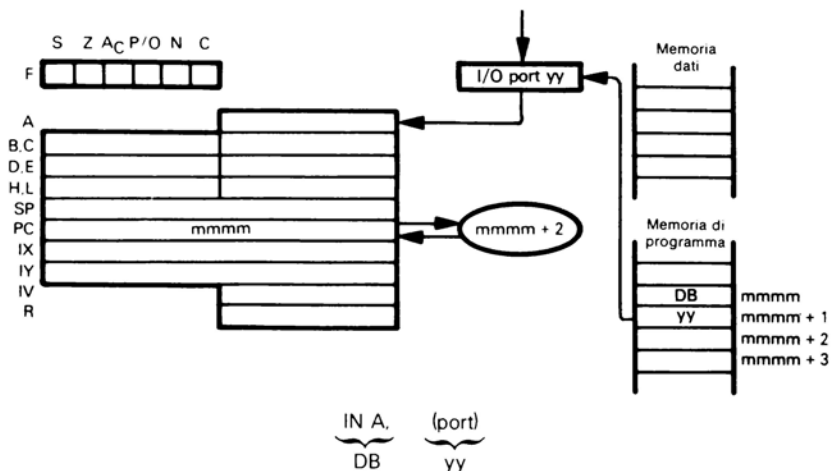
Questa istruzione mette la CPU nel modo 1 d'interruzione. In questo modo, la CPU risponde ad una interruzione eseguendo un restart (RST) alla locazione 0038_{16}.

IM 2 — INTERRUZIONE DI MODO 2

IM 2
ED 5E

Questa istruzione mette la CPU nel modo 2 d'interruzione. In questo modo, la CPU effettua una chiamata indiretta ad una locazione specificata in memoria. Si forma un indirizzo a 16 bit usando il contenuto del registro del Vettore d'Interruzione (IV) per gli otto bit superiori, mentre gli otto bit minori sono forniti dal dispositivo che interrompe. Ci si riferisca al Capitolo 5 per una descrizione completa dei modi di interruzione. Questa istruzione non influenza nessun registro o stato.

IN A,(port) — INGRESSO NELL'ACCUMULATORE

IN A, (port)
DB yy

Carica un byte dati nell'Accumulatore dalla porta di I/O (identificata dal secondo byte del codice oggetto dell'istruzione IN).

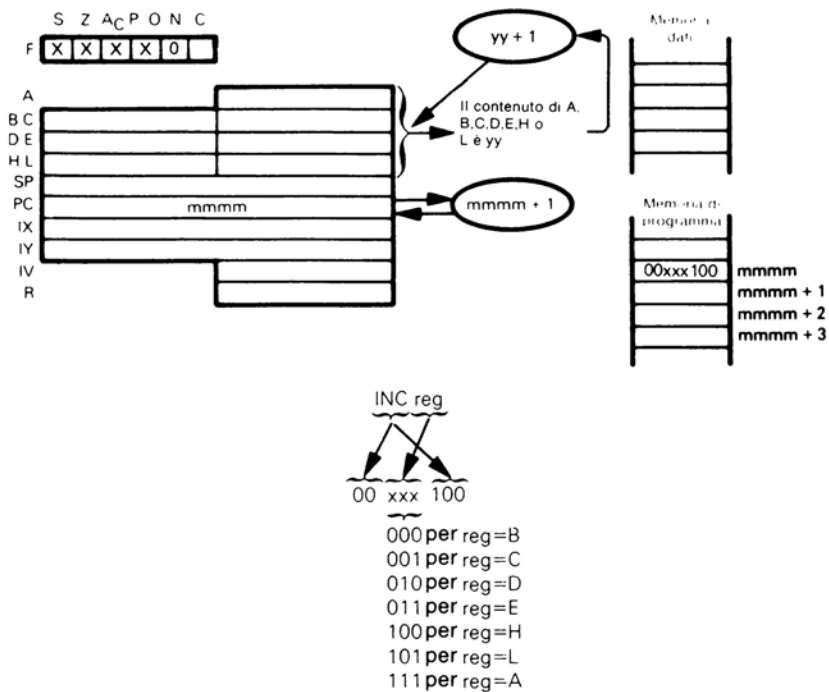
Supponiamo che 36_{16} sia contenuto nel buffer della porta di I/O $1A_{16}$. Dopo l'esecuzione dell'istruzione

IN A,(1AH)

l'Accumulatore conterrà 36_{16}.

L'istruzione IN non influenza nessuno stato.

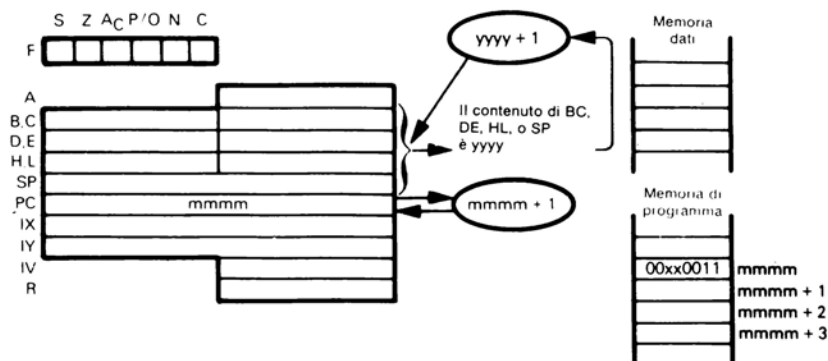
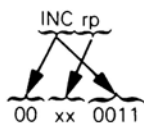
L'uso dell'istruzione IN è molto dipendente dall'hardware. Gli indirizzi validi per le porte di I/O sono determinati dal modo di implementazione della logica di I/O. E' pure possibile progettare un sistema a microcalcolatore che accede alla logica esterna usando istruzioni di riferimento alla memoria con specifici indirizzi di memoria.

INC reg — INCREMENTA IL CONTENUTO DEL REGISTRO

Somma 1 al contenuto del registro specificato.

Supponiamo che il Registro E contenga $A8_{16}$. Dopo l'esecuzione di

INC E

il registro E conterrà $A9_{16}$.

INC rp — INCREMENTA IL CONTENUTO DELLA COPPIA
INC IX   DI REGISTRI SPECIFICATA
INC IY

L'illustrazione mostra l'esecuzione di INC rp:

00 per rp è la coppia di registri BC
01 per rp è la coppia di registri DE
10 per rp è la coppia di registri HL
11 per rp è lo Stack Pointer

Somma 1 al valore di 16 bit contenuto nella coppia di registri specificata. Non sono influenzati i flag di stato.

Supponiamo che i registri D ed E contengano $2F7A_{16}$. Dopo l'esecuzione dell'istruzione

INC DE

i registri D ed E conterranno $2F7B_{16}$.

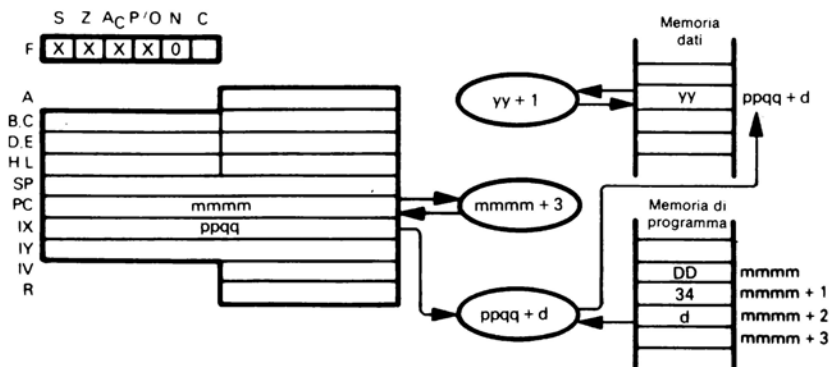
INC IX
DD 23

Somma 1 al valore di 16 bit contenuto nel registro IX.

INC IY
FD 23

Somma 1 al valore di 16 bit contenuto nel registro IY.

Così come DEC rp, DEC IX e DEX IY anche INC rp, INC IX e INC IY non influenzano nessun flag di stato. Questo è un difetto dell'insieme d'istruzioni dello Z80 ereditato dallo 8080.

INC (HL) — INCREMENTA IL CONTENUTO DELLA MEMORIA
INC (IX +disp)
INC (IY +disp)

L'illustrazione mostra l'esecuzione di INC (IX + d):

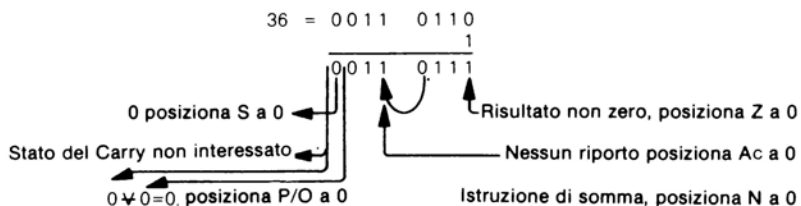
INC (IX+disp)

DD 34 d

Somma 1 al contenuto della locazione di memoria (specificata dalla somma del contenuto del Registro IX e del valore d del dislocamento).

Supponiamo che ppqq=4000_{16} e che la locazione di memoria $400F_{16}$ contenga 36_{16}. Dopo l'esecuzione dell'istruzione

INC (IX + 0FH)

la locazione di memoria $400F_{16}$ conterrà 37_{16}.

INC (IY+disp)

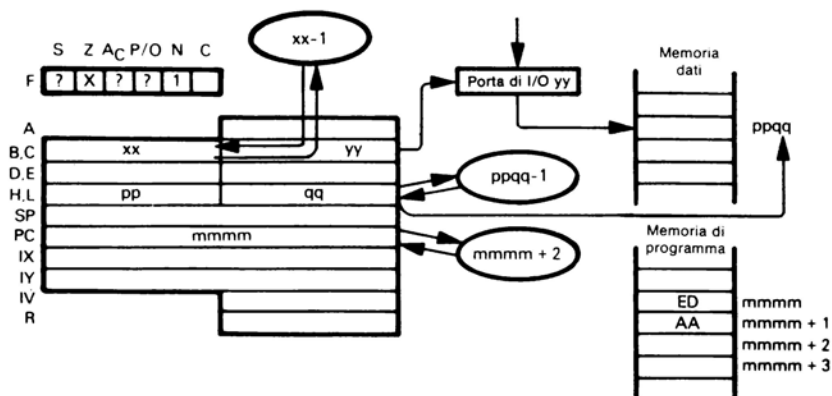
FD 34 d

Questa istruzione è identica a INC (IX + disp), tranne che essa usa il registro IY invece che il registro IX.

INC (HL)

34

Somma 1 al contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL).

IND — INGRESSO IN MEMORIA A DECREMENTO DEL PUNTATORE

$$\underbrace{\text{ED AA}}_{}$$

IND

ED AA

Ingresso di una porta di I/O (indirizzata dal Registro C) in una locazione di memoria (specificata da HL). Decremento dei Registri B e HL.

Supponiamo che xx=05_{16}, yy=15_{16}, ppqq=2400_{16} e che nel buffer della porta di I/O 15_{16} ci sia 19_{16}. Dopo l'esecuzione dell'istruzione

IND

la locazione di memoria 2400_{16} conterrà 19_{16}. Il registro B conterrà 04_{16} e la coppia di registri HL conterrà $23FF_{16}$.

INDR — INGRESSO IN MEMORIA E DECREMENTO DEL PUNTATORE FINCHE' IL CONTATORE DEI BYTE NON SIA ZERO

INDR
ED BA

INDR è identica a IND, ma si ripete finchè il Registro B=0.

Supponiamo che il Registro B contenga 03_{16}, che il Registro C contenga 15_{16} e che HL contenga 2400_{16}. La seguente sequenza di byte sia disponibile alla porta di I/O 15_{16}:

17_{16}, 59_{16} e AE_{16}

Dopo l'esecuzione di

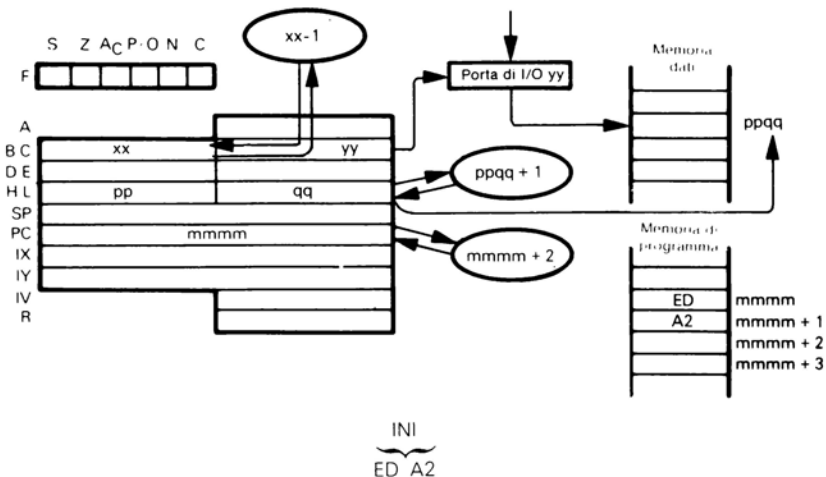
INDR

la coppia di registri HL conterrà $23FD_{16}$ e il Registro B conterrà zero e le locazioni di memoria avranno i contenuti seguenti:

Locazione	Contenuto
2400	17_{16}
23FF	59_{16}
23FE	AE_{16}

Questa istruzione è estremamente utile per caricare blocchi di dati da un dispositivo di ingresso in memoria.

INI — INGRESSO IN MEMORIA E INCREMENTO DEL PUNTATORE

INI
ED A2

Ingresso da una porta di I/O (indirizzata dal Registro C) in una locazione di memoria (specificata da HL). Decremento del Registro B; incremento della coppia di registri HL.

Supponiamo che xx=05_{16}, yy=15_{16}, ppqq=2400_{16} e che nel buffer della porta di I/O 15_{16} ci sia 19_{16}. Dopo l'esecuzione dell'istruzione

INI

la locazione di memoria 2400_{16} conterrà 19_{16}. Il registro B conterrà 04_{16} e la coppia di registri HL conterrà 2401_{16}.

INIR — INGRESSO IN MEMORIA ED INCREMENTO DEL PUNTATORE FINCHE' IL CONTATORE DEI BYTE NON SIA ZERO

$$\underbrace{\text{INIR}}_{\text{ED B2}}$$

INIR è identico a INI, ma si ripete finchè il Registro B=0.

Supponiamo che il Registro B contenga 03_{16}, che il Registro C contenga 15_{16} e che HL contenga 2400_{16}. La seguente sequenza di byte sia disponibile alla porta di I/O 15_{16}:

17_{16}, 59_{16} e AE_{16}

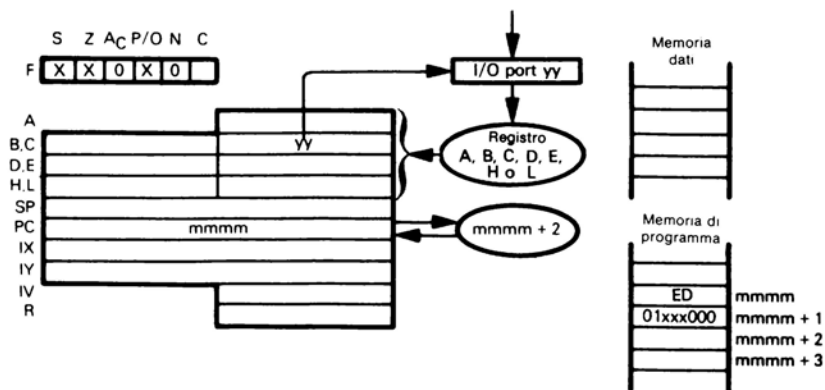
Dopo l'esecuzione di

INIR

la coppia di registri HL conterrà 2403_{16} ed il Registro B conterrà zero e le locazioni di memoria avranno i contenuti seguenti:

Locazione	Contenuto
2400	17_{16}
2401	59_{16}
2402	AE_{16}

Questa istruzione è estremamente utile per caricare blocchi di dati da un dispositivo in memoria.

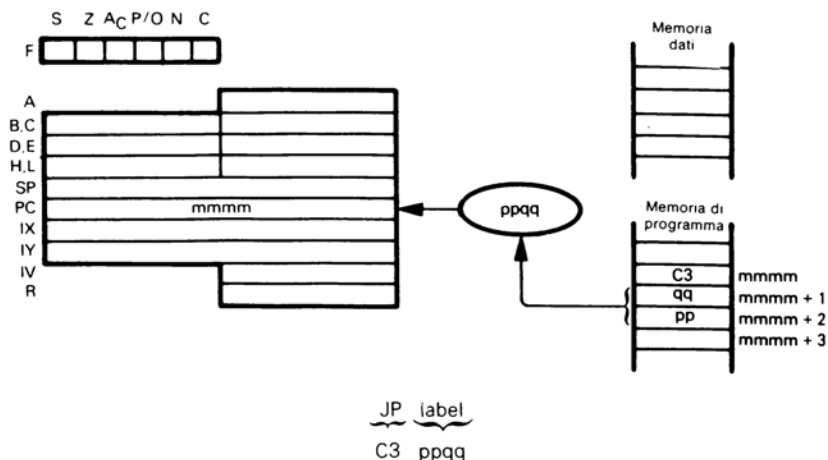
IN reg,(C) — INGRESSO IN UN REGISTRO

Carica un byte di dato nel registro specificato (reg) dalla porta di I/O (identificata dal contenuto del registro C).

Supponiamo che 42_{16} sia contenuto nel buffer della porta di I/O 36_{16} e che il Registro C contenga 36_{16}. Dopo l'esecuzione dell'istruzione

IN D,(C)

il registro D conterrà 42_{16}.

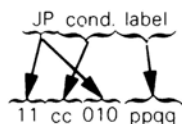
Durante l'esecuzione dell'istruzione, il contenuto del Registro B è posto sulla metà superiore del Bus degli Indirizzi, rendendolo capace di estendere il numero delle porte di I/O indirizzabili.

JP label — SALTA ALL'ISTRUZIONE IDENTIFICATA NELL'OPERANDO

Carica il contenuto del secondo e del terzo byte del codice oggetto dell'istruzione JP nel Contatore dei Programmi; questo diventa l'indirizzo di memoria della prossima istruzione da eseguire. Il contenuto precedente del Contatore dei Programmi è perduto.

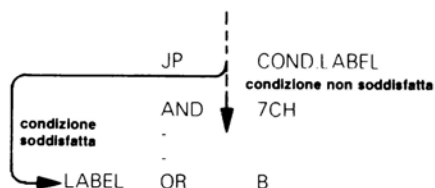
Nella sequenza seguente:

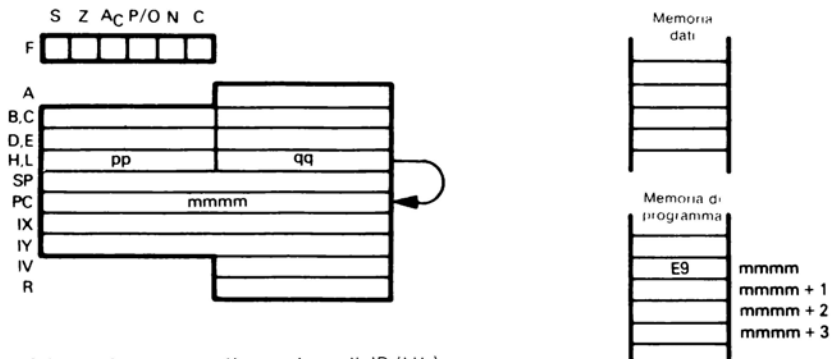
```
            JP    NEXT
            AND   7FH
            —
            —
NEXT        CPL
```

L'istruzione CPL sarà eseguita dopo l'esecuzione dell'istruzione JP. L'istruzione AND non sarà mai eseguita, a meno che in qualche altro posto della sequenza delle istruzioni una istruzione JP non salti a questa istruzione.

JP condition, label — SALTA ALL'INDIRIZZO IDENTIFICATO NELL'OPERANDO SE LA CONDIZIONE E' SODDISFATTA

		Condizione	Flag pertinente
000	NZ	Non-Zero	Z
001	Z	Zero	Z
010	NC	Nessun Carry	C
011	C	Carry	C
100	PO	Parità dispari	P/O
101	PE	Parità pari	P/O
110	P	Segno positivo	S
111	M	Segno negativo	S

Questa istruzione è identica all'istruzione JP, tranne che il salto sarà effettuato solo se la condizione è soddisfatta; altrimenti, si eseguirà l'istruzione seguente in sequenza l'istruzione JP condition.

Consideriamo la sequenza di istruzioni:

Dopo l'esecuzione dell'istruzione JP cond,label, se la condizione è soddisfatta si eseguirà allora l'istruzione OR. Se la condizione non è soddisfatta, si eseguirà la istruzione AND, che è l'istruzione successiva della sequenza.

JP (HL) — SALTO ALL'INDIRIZZO SPECIFICATO DAL CONTENUTO
JP (IX) DEL REGISTRO A 16 BIT
JP (IY)

L'illustrazione mostra l'esecuzione di JP (HL):

```
JP (HL)
  E9
```

Il contenuto della coppia di registri HL è messo nel Contatore di Programma: si realizza, quindi, un salto con indirizzamento implicito.

La sequenza d'istruzioni

```
LD   H,ADDR
JP   (HL)
```

ha esattamente lo stesso effetto netto dell'istruzione singola

```
JP   ADDR
```

Entrambe specificano che successivamente si deve eseguire l'istruzione avente come label ADDR.

L'istruzione JP (HL) è utile quando si vuole incrementare un indirizzo di ritorno per un sottoprogramma che ha ritorni multipli.

Consideriamo la seguente chiamata al sottoprogramma SUB:

```
CALL SUB        ; Chiamata del sottoprogramma
JP   ERR        ; Ritorno errore
                ; Ritorno buono
```

Usando RET per tornare da SUB si tornerebbe all'esecuzione di JP ERR; perciò, se si esegue SUB senza condizioni che rivelino errori, si ritorna come segue:

```
POP  HL         ; Estrae l'indirizzo di ritorno per HL
INC  HL         ; Somma tre all'indirizzo di ritorno
INC  HL
INC  HL
JP   (HL)       ; Ritorno
```

```
JP (IX)
 DD E9
```

Questa istruzione è identica all'istruzione JP (HL), tranne che essa usa il registro IX invece della coppia di registri HL.

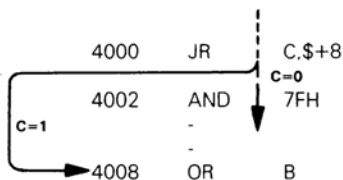
JP (IY)
FD E9

Questa istruzione è identica all'istruzione JP (HL), tranne che essa usa il registro IY invece della coppia di registri HL.

JR C,disp — SALTO RELATIVO AL CONTENUTO DEL CONTATORE DI PROGRAMMA SE IL CARRY E' POSTO AD 1

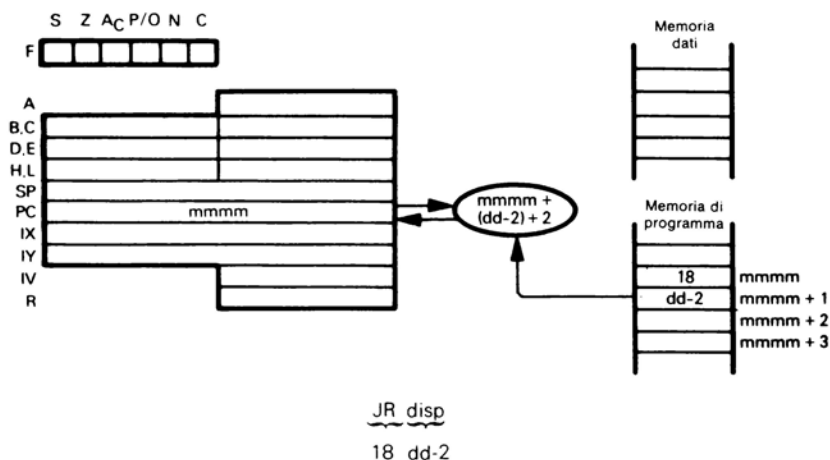
JR C, disp
38 dd-2

Questa istruzione è identica all'istruzione JR disp, tranne che si esegue il salto solo se lo stato di Carry è uguale a 1; altrimenti, si esegue la istruzione successiva.

Nella seguente sequenza di istruzioni:

Dopo l'istruzione JR C,$ + 8, si esegue l'istruzione OR se lo stato di Carry è uguale a 1. Si esegue l'istruzione AND se lo stato di Carry è uguale a 0.

JR disp — SALTO RELATIVO AL CONTENUTO PRESENTE NEL CONTATORE DI PROGRAMMA

JR disp

18 dd-2

Somma il contenuto del secondo byte del codice oggetto dell'istruzione JR, il contenuto del Contatore di Programma e 2. Carica la somma nel Contatore di Programma. Il salto è misurato dall'indirizzo del codice operativo dell'istruzione, ed ha un campo variabile da −126 a +129 byte. L'Assembler si regola automaticamente per il doppio incremento di PC.

La seguente affermazione in linguaggio assembly è usata per saltare quattro passi avanti l'indirizzo 4000_{16}.

JR $ + 4

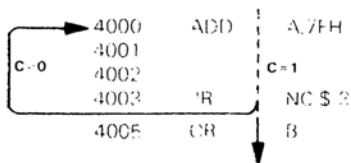
Il risultato di questa istruzione è il seguente:

Locazione	Istruzione
4000	18
4001	02
4002	-
4003	
4004	- ◄—— nuovo valore di PC

JR NC,disp — SALTO RELATIVO AL CONTENUTO DEL CONTATORE DI PROGRAMMA SE IL FLAG DI CARRY E' AZZERATO

JR NC,disp
30 dd-2

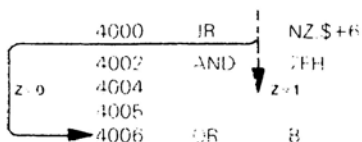
Questa istruzione è identica all'istruzione JR disp, tranne che il salto è eseguito solo se lo stato di Carry è uguale a 0; altrimenti si esegue l'istruzione successiva.

Nella seguente sequenza di istruzioni:

Dopo, l'istruzione JR NC,$ – 3, si esegue l'istruzione OR se lo stato di Carry è uguale ad 1. Si esegue l'istruzione ADD se lo stato di Carry è uguale a 0.

JR NZ,disp — SALTO RELATIVO AL CONTENUTO DEL CONTATORE DI PROGRAMMA SE IL FLAG ZERO E' AZZERATO

JR NZ,disp
20 dd-2

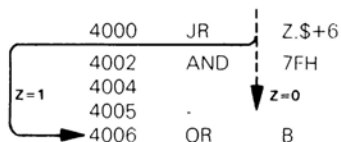
Questa istruzione è identica all'istruzione JR disp, tranne che il salto è eseguito solo se lo stato Zero è uguale a 0; altrimenti si esegue l'istruzione successiva.

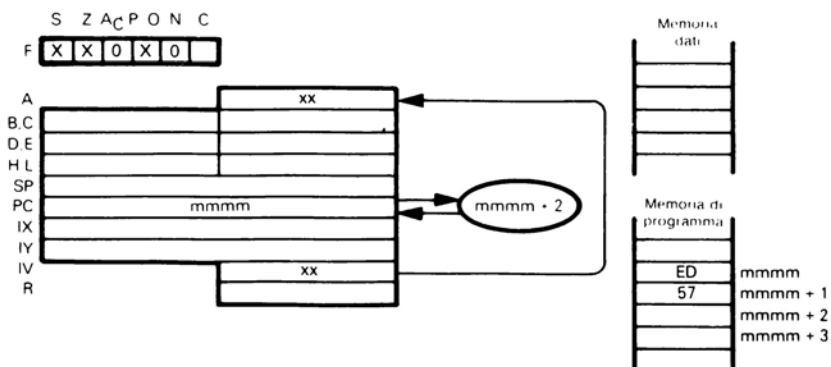
Nella seguente sequenza di istruzioni:

dopo l'istruzione JR NZ,$ + 6, si esegue l'istruzione OR se lo stato di Zero è uguale a 0. Si esegue l'istruzione AND se lo stato Zero è uguale a 1.

JR Z,disp — SALTO RELATIVO AL CONTENUTO DEL CONTATORE DI PROGRAMMA SE IL FLAG ZERO E' POSIZIONATO AD 1

JR Z,disp
28 dd-2

Questa istruzione è identica all'istruzione JR disp, tranne che il salto è eseguito solo se lo stato Zero è uguale ad 1; altrimenti, si esegue l'istruzione successiva.

Nella seguente sequenza di istruzioni:

Dopo l'istruzione JR Z,$ + 6, si esegue l'istruzione OR se lo stato Zero è uguale a 1. Si esegue l'istruzione AND se lo stato Zero è uguale a 0.

LD A,IV — SPOSTA IL CONTENUTO DEL VETTORE
LD A,R — D'INTERRUZIONE O DEL REGISTRO DI RINFRESCO NELL'ACCUMULATORE

L'illustrazione mostra l'esecuzione di LD A,IV:

LD A,IV
ED 57

Sposta il contenuto del registro del Vettore d'Interruzione nell'Accumulatore e riflette lo stato di abilitazione dell'interruzione nel flag di Parità/Overflow.

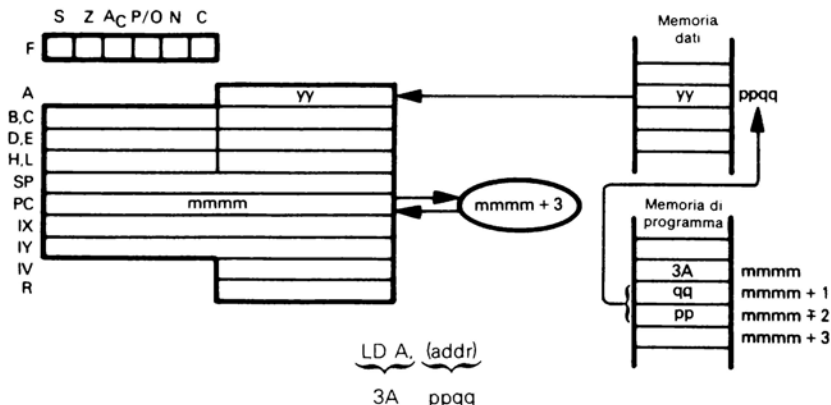
Supponiamo che il registro del Vettore d'Interruzione contenga $7F_{16}$ e che le interruzioni siano disabilitate. Dopo l'esecuzione di

LD A,IV

Il Registro A conterrà $7F_{16}$ e P/O sarà 0.

LD A.R
ED 5F

Sposta il contenuto del registro di Rinfresco nell'Accumulatore. Il valore del flip-flop d'interruzione apparirà nel flag di Parità/Overflow.

LD A,(addr) — CARICA L'ACCUMULATORE DALLA MEMORIA USANDO UN INDIRIZZAMENTO DIRETTO

Carica il contenuto del byte di memoria (indirizzato direttamente dal secondo e dal terzo byte del codice oggetto dell'istruzione LD A,(addr)) nell'Accumulatore. Supponiamo che il byte di memoria $084A_{16}$ contenga 20_{16}. Dopo l'esecuzione dell'istruzione

```
label   EQU  084AH
—
—
—
LD      A,(label)
```

l'Accumulatore conterrà 20_{16}.

Ricordiamo che EQU è una direttiva Assembler piuttosto che un'istruzione; essa dice all'Assembler di usare il valore a 16 bit $084A_{16}$ dovunque compaia label.

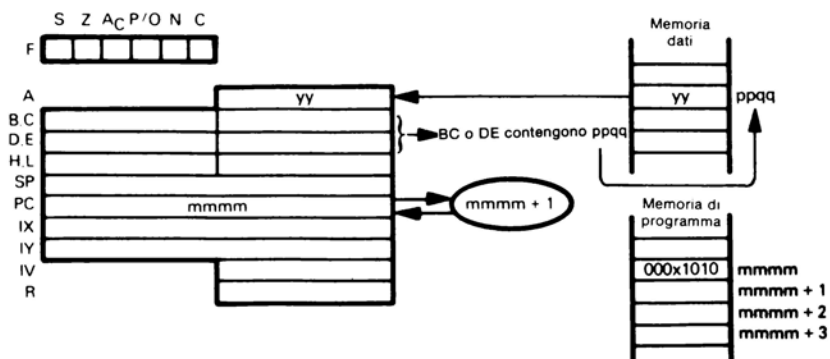
L'istruzione

```
LD      A,(label)
```

è equivalente alle due istruzioni

```
LD      HL,label
LD      A,(HL)
```

Quando si carica un singolo valore dalla memoria, si preferisce l'istruzione LD A,(label): si usano una sola istruzione a tre byte di programma oggetto per fare ciò che la combinazione LD HL,label, LD A,(HL) fa con due istruzioni e con quattro byte di programma oggetto. Inoltre, la combinazione LD HL,label e LD A,(HL) usa i registri H ed L che non sono usati da LD A,(label).

LD A,(rp) — CARICA L'ACCUMULATORE DALLA LOCAZIONE DI MEMORIA INDIRIZZATA DALLA COPPIA DI REGISTRI

Carica il contenuto del byte di memoria (indirizzato dalla coppia di registri BC o DE) nell'Accumulatore.

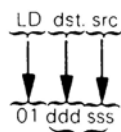
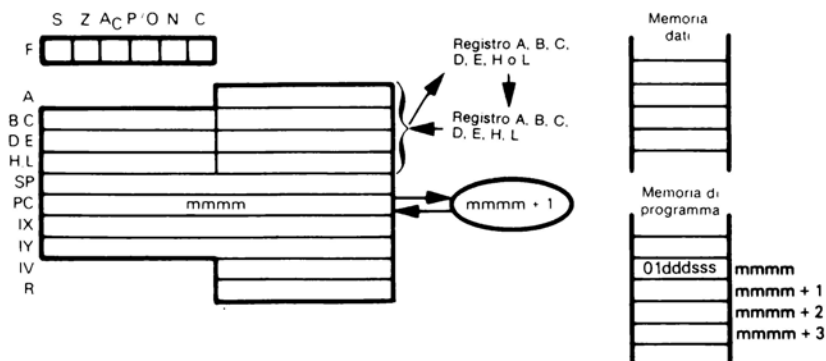
Supponiamo che il registro B contenga 08_{16}, che il registro C contenga $4A_{16}$ e che il byte di memoria $084A_{16}$ contenga $3A_{16}$. Dopo l'esecuzione dell'istruzione

LD A,(BC)

l'Accumulatore conterrà $3A_{16}$.

Normalmente, si useranno insieme LD A,(rp) e LD rp,data, poichè l'istruzione LD rp, data carica un indirizzo di 16 bit nei registri BC o DE come segue:

```
LD    BC,084AH
LD    A,(BC)
```

LD dst,src — SPOSTA IL CONTENUTO DEL REGISTRO SORGENTE NEL REGISTRO DI DESTINAZIONE

Il contenuto di un qualsiasi registro indicato è caricato in un altro registro.

Per esempio:

LD A,B

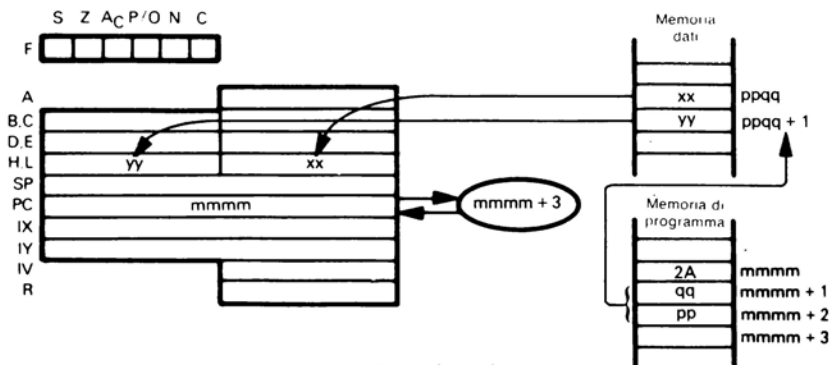
Carica il contenuto del Registro B nel Registro A.

LD L,D

carica il contenuto del Registro D nel Registro L.

LD C,C

non fa niente, poichè il registro C è stato specificato sia come sorgente che come destinazione.

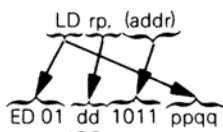
LD HL,(addr) — CARICA LA COPPIA DI REGISTRI OPPURE
LD rp,(addr) IL REGISTRO INDICE DALLA MEMORIA USANDO
LD IX,(addr) UN INDIRIZZAMENTO DIRETTO
LD IY,(addr)

L'illustrazione mostra l'esecuzione di LD HL,(ppqq):

LD HL,addr
2A ppqq

Carica la coppia di registri HL dalla locazione di memoria indirizzata direttamente.

Supponiamo che la locazione di memoria 4004_{16} contenga ADD_{16} e che la locazione di memoria 4005_{16} contenga 12_{16}. Dopo l'esecuzione dell'istruzione

LD HL,(4004H)

la coppia di registri HL conterrà $12AD_{16}$.

00 per rp è la coppia di registri BC
01 per rp è la coppia di registri DE
10 per rp è la coppia di registri HL
11 per rp è lo Stack Pointer

Carica la coppia di registri dalla memoria indirizzata direttamente.

Supponiamo che la locazione di memoria $49FF_{16}$ contenga BE_{16} e che la locazione di memoria $4A00_{16}$ contenga 33_{16}. Dopo l'esecuzione dell'istruzione

LD DE,(49FFH)

la coppia di registri DE conterrà $33BE_{16}$.

LD IX,(addr)
DD 2A ppqq

Carica il registro IX dalla memoria indirizzata direttamente.

Supponiamo che la locazione di memoria $D111_{16}$ contenga FF_{16} e che la locazione di memoria $D112_{16}$ contenga 56_{16}. Dopo l'esecuzione dell'istruzione

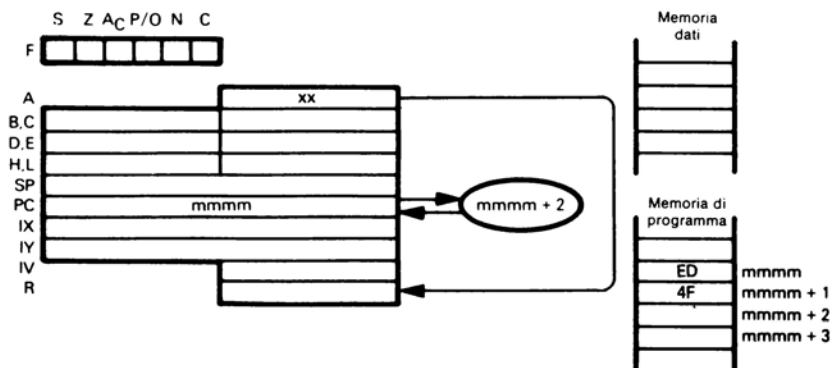
LD IX,(D111H)

il registro IX conterrà $56FF_{16}$.

LD IY,(addr)
FD 2A ppqq

Carica il registro IY dalla memoria indirizzata direttamente.

Interessa il registro IY invece di IX. Altrimenti è indicata a LD IX,(addr).

LD IV,A – CARICA IL VETTORE D'INTERRUZIONE OPPURE
LD R,A IL REGISTRO DI RINFRESCO DELL'ACCUMULATORE

L'illustrazione mostra l'esecuzione di LD R,A:

LD R,A
ED 4F

Carica il registro di Rinfresco dall'Accumulatore.

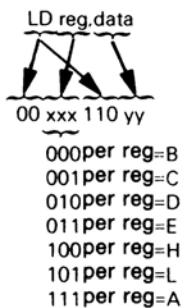
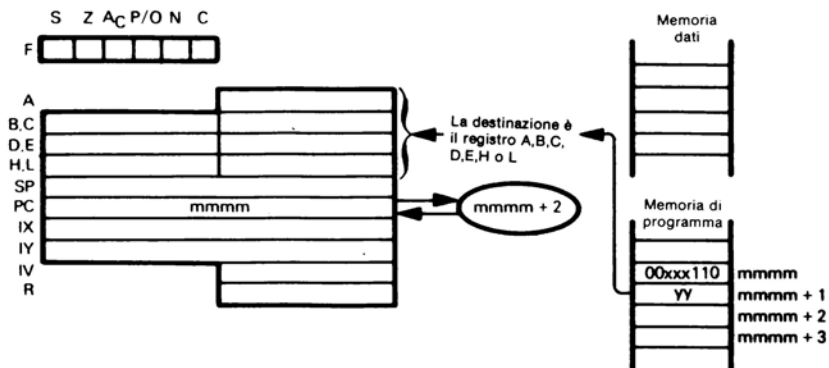
Supponiamo che l'Accumulatore contenga $7F_{16}$. Dopo l'esecuzione dell'Istruzione

LD R,A

il registro di Rinfresco conterrà $7F_{16}$.

LD IV,A
ED 47

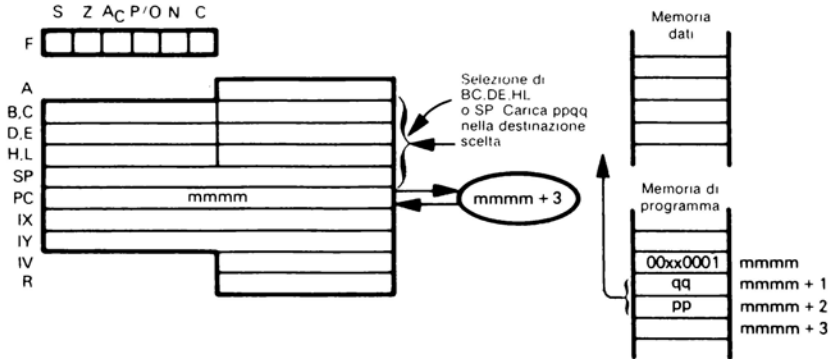
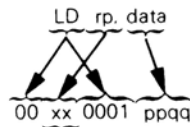
Carica il registro del Vettore d'Interruzione dall'Accumulatore.

LD reg,data — CARICA IMMEDIATAMENTE NEL REGISTRO

Carica il contenuto del secondo byte del codice oggetto in uno dei registri.

Dopo che si è eseguita l'istruzione

```
LD A,2AH
```

si è caricato $2A_{16}$ nell'Accumulatore.

LD rp,data — CARICA UN DATO IMMEDIATO DI 16 BIT
LD IX,data NEL REGISTRO
LD IY,data

L'illustrazione mostra l'esecuzione di LD rp,data:

00 per rp è la coppia di registri BC
01 per rp è la coppia di registri DE
10 per rp è la coppia di registri HL
11 per rp è lo Stack Pointer

Carica il contenuto del secondo e del terzo byte del codice oggetto nella coppia di registri selezionata. Dopo l'esecuzione dell'istruzione

LD SP,217AH

lo Stack Pointer conterrà $217A_{16}$.

LD IX, data
DD 21 ppqq

Carica il contenuto del secondo e del terzo byte del codice oggetto nel registro Indice IX.

LD IY, data
FD 21 ppqq

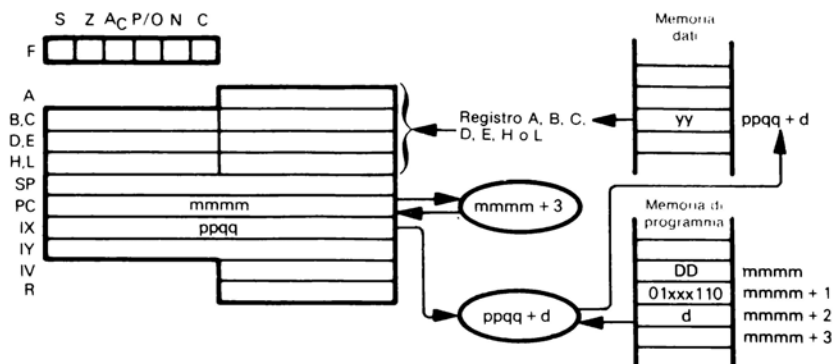
Carica il contenuto del secondo e del terzo byte del codice oggetto nel registro Indice IY. E' da notare che l'istruzione LD rp,data è equivalente a due istruzioni LD reg,data.

Per esempio:

```
LD      HL,032AH
```

è equivalente a

```
LD      H,03H
LD      L,2AH
```

LD reg,(HL) — CARICA IL REGISTRO DALLA MEMORIA
LD reg,(IX +disp)
LD reg,(IY +disp)

L'illustrazione mostra l'esecuzione di LD reg,(IX + disp).

Carica il registro specificato dalla locazione di memoria (specificata dalla somma del contenuto del registro IX e del peso d del dislocamento).

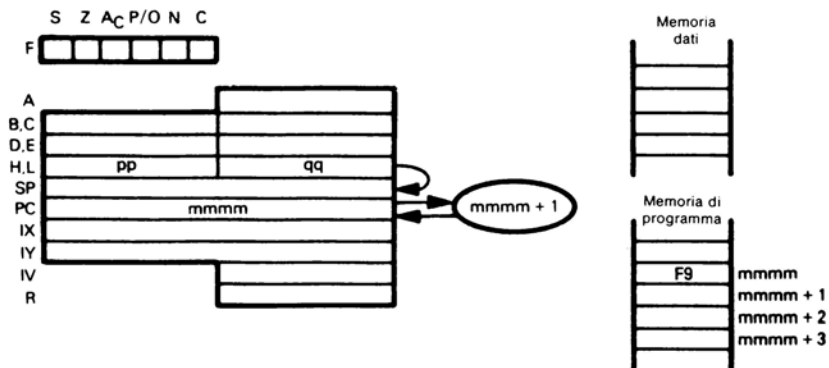
Supponiamo che ppqq=4004_{16} e che la locazione di memoria 4010_{16} contenga FF_{16}. Dopo l'esecuzione dell'istruzione

LD B(IX + 0CH)

il registro B conterrà FF_{16}.

Questa istruzione è identica a LD reg,(IX + disp), tranne che essa usa il registro IY invece del registro IX.

Carica il registro specificato dalla locazione di memoria (specificata dal contenuto della coppia di registri HL).

LD SP,HL — SPOSTA IL CONTENUTO DI HL OPPURE DEL REGISTRO
LD SP,IX INDICE NELLO STACK POINTER
LD SP,IY

L'illustrazione mostra l'esecuzione di LD SP,HL:

LD SP,HL
F9

Carica il contenuto di HL nello Stack Pointer.

Supponiamo che pp=08_{16} e che qq=$3F_{16}$. Dopo l'esecuzione dell'istruzione

LD SP,HL

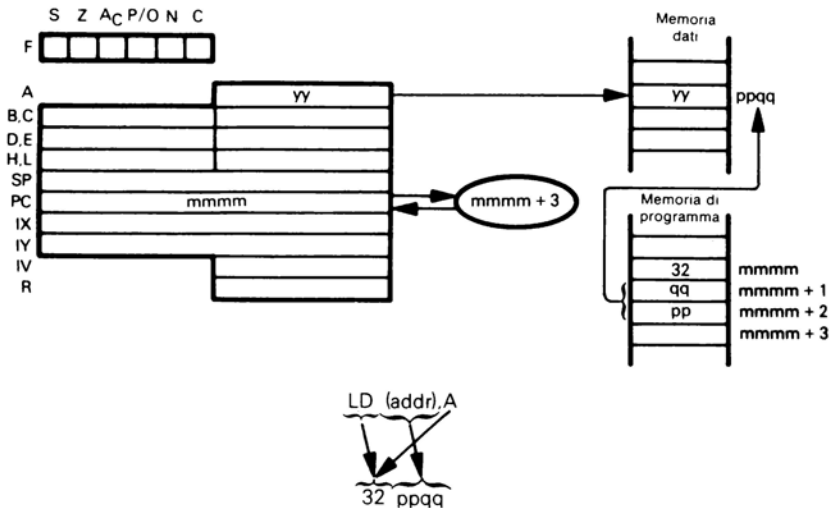
lo Stack Pointer conterrà $083F_{16}$.

LD SP,IX
DD F9

Carica il contenuto del Registro Indice IX nello Stack Pointer.

LD SP,IY
FD F9

Carica il contenuto del Registro Indice IY nello Stack Pointer.

LD (addr),A — MEMORIZZA L'ACCUMULATORE NELLA MEMORIA USANDO UN INDIRIZZAMENTO DIRETTO

```
LD (addr),A

32 ppqq
```

Immagazzina il contenuto dell'Accumulatore nel byte di memoria indirizzato direttamente dal secondo e dal terzo byte del codice oggetto dell'istruzione LD (addr),A.

Supponiamo che l'Accumulatore contenga $3A_{16}$. Dopo l'esecuzione dell'istruzione

```
label  EQU  084AH
       —
       —
       LD   (label),A
```

il byte di memoria $084A_{16}$ conterrà $3A_{16}$.

Ricordiamo che EQU è una direttiva Assembler piuttosto che un'istruzione; essa dice all'Assembler di usare il valore a 16 bit 084AH ogni volta che appare la parola "label".

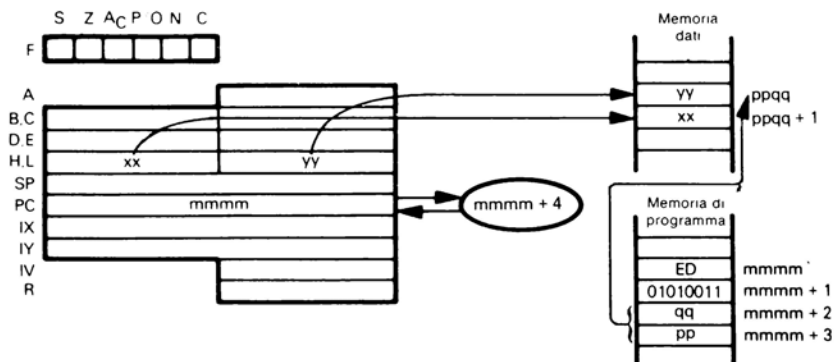
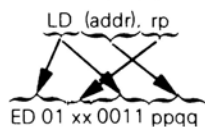
L'istruzione

```
LD (addr),A
```

è equivalente alle due istruzioni

```
LD H,label
LD (HL),A
```

Quando si immagazzina in memoria un singolo valore dati, si preferisce l'istruzione LD (label),A, perchè essa usa una sola istruzione e tre byte di programma oggetto per fare ciò che la combinazione LD H,label ed LD (HL),A fa in due istruzioni e in quattro byte di programma oggetto. Inoltre, la combinazione LD H,label ed LD (HL),A usa i registri H ed L, mentre l'istruzione LD (label),A no.

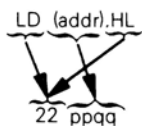
LD (addr),HL – IMMAGAZZINA LA COPPIA DI REGISTRI
LD (addr),rp   O IL REGISTRO INDICE IN MEMORIA USANDO
LD (addr),xy   UN INDIRIZZAMENTO DIRETTO

L'illustrazione mostra l'esecuzione di LD (ppqq),DE

00 per rp è la coppia di registri BC
01 per rp è la coppia di registri DE
10 per rp è la coppia di registri HL
11 per rp è lo Stack Pointer

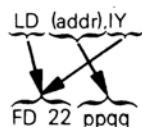
Immagazzina in memoria il contenuto della coppia di registri specificata. Il terzo e il quarto byte del codice oggetto danno l'indirizzo della locazione di memoria dove si deve scrivere il byte di ordine minore. Il byte di ordine maggiore è scritto nella locazione di memoria successiva della sequenza.

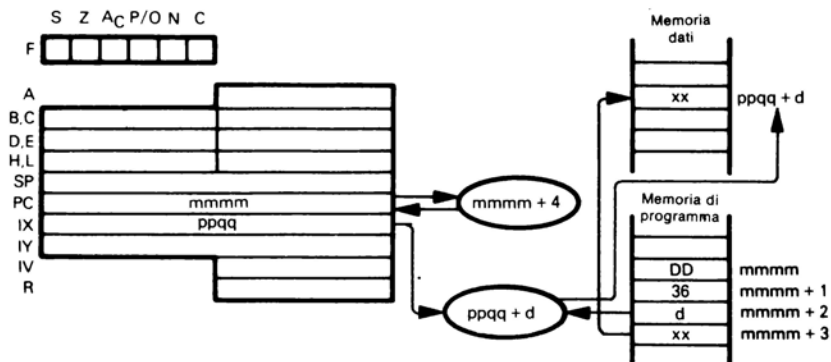
Supponiamo che la coppia di registri BC contenga $3C2A_{16}$. Dopo l'esecuzione della istruzione

```
label   EQU   084AH
–
–
–
LD      (label),BC
```

il byte di memoria $084A_{16}$ conterrà $2A_{16}$. Il byte di memoria $084B_{16}$ conterrà $3C_{16}$.

Ricordiamo che EQU è una direttiva Assembler piuttosto che un'istruzione; essa dice all'Assembler di usare il valore a 16 bit $084A_{16}$ ogni volta che appare la parola "label".

Questa è una versione a tre byte di LD (addr),rp che specifica direttamente HL come coppia di registri sorgente.

Immagazzina il contenuto del registro Indice IX in memoria. Il terzo e il quarto byte del codice oggetto danno l'indirizzo della locazione di memoria dove si deve scrivere il byte di ordine minore. Il byte di ordine maggiore è scritto nella locazione di memoria successiva della sequenza.

Questa istruzione è identica all'istruzione LD (addr),IX, tranne che essa usa il registro IY invece del registro IX.

LD (HL),data — CARICA IMMEDIATAMENTE NELLA MEMORIA
LD (IX + disp), data
LD (IY + disp), data

L'illustrazione mostra l'esecuzione di LD (IX + d),xx:

LD (IX+disp),data
DD 36 d xx

Carica immediatamente nella locazione di memoria indicata dall'indirizzamento relativo alla base.

Supponiamo che ppqq=5400_{16}. Dopo l'esecuzione dell'istruzione

LD (IX + 9),FAH

la locazione di memoria 5409_{16} conterrà FA_{16}.

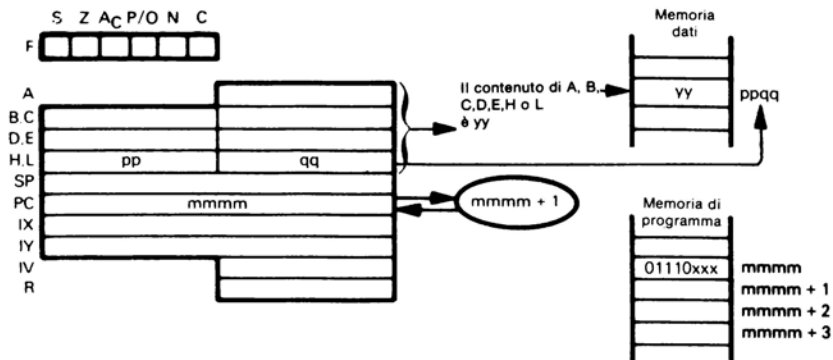
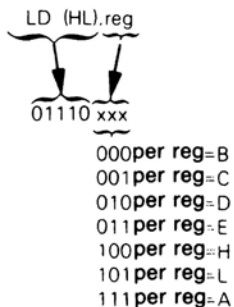
LD (IY+disp),data
FD 36 d xx

Questa istruzione è identica a LD (IX + disp),data, ma essa usa il registro IY invece del registro IX.

LD (HL),data
36 xx

Carica immediatamente nella locazione di memoria (specificata dal contenuto della coppia di registri HL).

Le istruzioni di Caricamento Immediato in Memoria sono molto meno usate delle istruzioni di Caricamento Immediato in Registri.

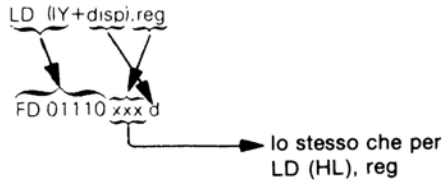
LD (HL),reg — CARICA LA MEMORIA DA UN REGISTRO
LD (IX +disp),reg
LD (IY +disp),reg

L'illustrazione mostra l'esecuzione di LD (HL),reg:

Carica la locazione di memoria (specificata dal contenuto della coppia di registri HL) dal registro specificato.

Supponiamo che ppqq=4500_{16} e che il Registro C contenga $F9_{16}$. Dopo l'esecuzione dell'istruzione

LD (HL),C

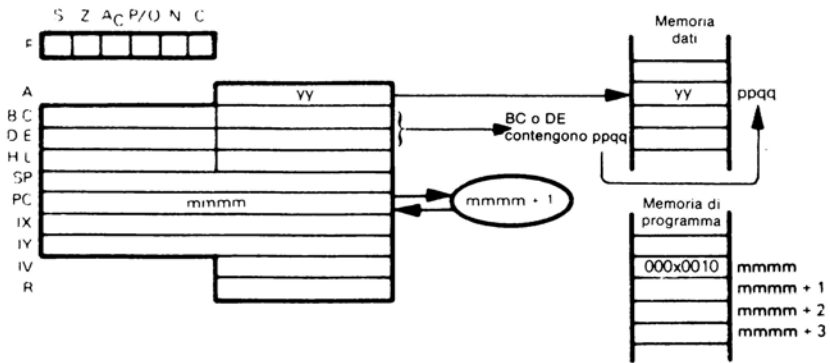
la locazione di memoria 4500_{16} conterrà $F9_{16}$.

Carica la locazione di memoria (specificata dalla somma del contenuto del registro IX

e del valore d del dislocamento) dal registro specificato.

Questa istruzione è identica a LD (IX + disp),reg, tranne che essa usa il registro IY invece del registro IX.

LD (rp),A – CARICA L'ACCUMULATORE NELLA LOCAZIONE DI MEMORIA INDIRIZZATA DALLA COPPIA DI REGISTRI

LD (rp),A

000 x 0010

0 se la coppia di registri = BC
1 se la coppia di registri = DE

Immagazzina l'Accumulatore nel byte di memoria indirizzato dalla coppia di registri BC o DE.

Supponiamo che la coppia di registri BC contenga $084A_{16}$ e che l'Accumulatore contenga $3A_{16}$. Dopo l'esecuzione dell'istruzione

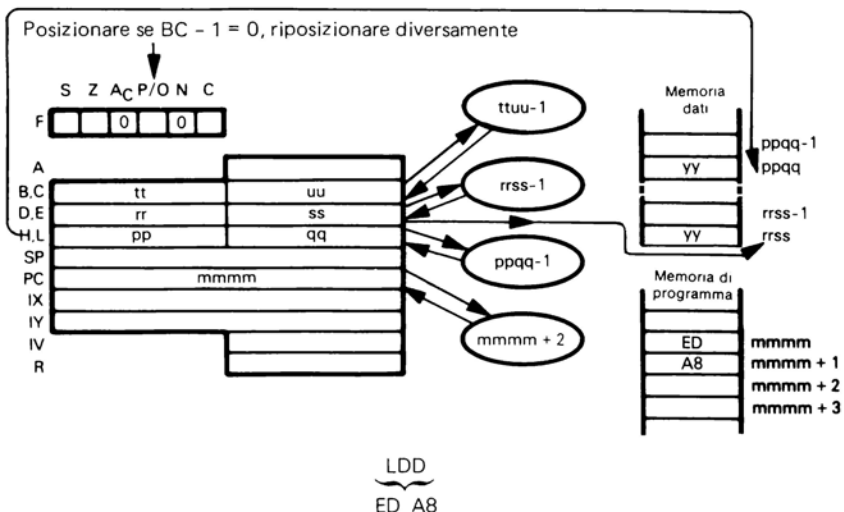
LD (BC),A

il byte di memoria $084A_{16}$ conterrà $3A_{16}$.

Le istruzioni LD (rp),A e LD rp,data saranno normalmente usate insieme, poichè l'istruzione LD rp,data carica un indirizzo a 16 bit nei registri BC o DE come segue:

```
LD BC,084AH
LD (BC),A
```

LDD — TRASFERISCE DATI TRA LOCAZIONI DI MEMORIA. DECREMENTA GLI INDIRIZZI DELLA DESTINAZIONE E DELLA SORGENTE

$$\underbrace{\text{LDD}}_{\text{ED A8}}$$

Trasferisce un byte del dato dalla locazione di memoria indirizzata dalla coppia di registri HL alla locazione di memoria indirizzata dalla coppia di registri DE. Decrementa il contenuto della coppia di registri BC, DE e HL.

Supponiamo che la coppia di registri BC contenga $004F_{16}$, che DE contenga 4545_{16}, che HL contenga 2012_{16} e che la locazione di memoria 2012_{16} contenga 18_{16}. Dopo l'esecuzione dell'istruzione

LDD

la locazione di memoria 4545_{16} conterrà 18_{16}, la coppia di registri BC conterrà $004E_{16}$, DE conterrà 4544_{16} e HL conterrà 2011_{16}.

LDDR — TRASFERISCE DATI TRA LOCAZIONI DI MEMORIA FINCHE' IL CONTATORE DEI BYTE NON SIA ZERO. DECREMENTA GLI INDIRIZZI DELLA DESTINAZIONE E DELLA SORGENTE

LDDR

ED B8

Questa istruzione è identica a LDD, tranne che essa si ripete finchè la coppia di registri BC non contenga zero. Dopo ogni trasferimento di un dato, si riconosceranno le interruzioni e si eseguiranno due cicli di rinfresco.

Supponiamo di avere i seguenti contenuti in memoria e nelle coppie di registri:

Registro/Contenuto		Locazione/Contenuto	
HL	2012_{16}	2012_{16}	18_{16}
DE	4545_{16}	2011_{16}	AA_{16}
BC	0003_{16}	2010_{16}	25_{16}

Dopo l'esecuzione di

LDDR

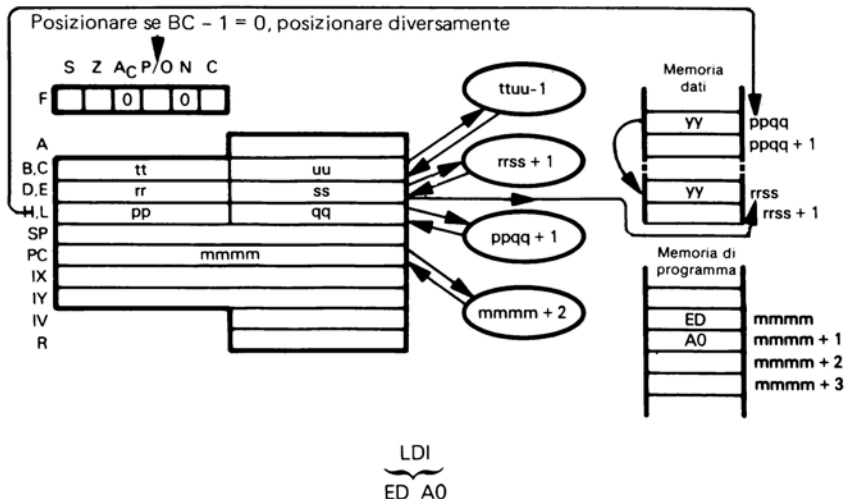
le coppie di registri e le locazioni di memoria avranno i contenuti seguenti:

Registro/Contenuto		Locazione/Contenuto		Locazione/Contenuto	
HL	2009_{16}	2012_{16}	18_{16}	4545_{16}	18_{16}
DE	4542_{16}	2011_{16}	AA_{16}	4544_{16}	AA_{16}
BC	0000_{16}	2010_{16}	25_{16}	4543_{16}	25_{16}

Questa istruzione è estremamente utile per trasferire blocchi di dati da un'area di memoria ad un'altra.

LDI — TRASFERISCE DATI TRA LOCAZIONI DI MEMORIA. INCREMENTA GLI INDIRIZZI DELLA DESTINAZIONE E DELLA SORGENTE

LDI

ED A0

Trasferisce un byte dati dalla locazione di memoria indirizzata dalla coppia di registri HL nella locazione di memoria indirizzata dalla coppia di registri DE. Incrementa il contenuto delle coppie di registri HL e DE. Decrementa il contenuto della coppia di registri BC.

Supponiamo che la coppia di registri BC contenga $004F_{16}$, che DE contenga 4545_{16}, che HL contenga 2012_{16} e che la locazione di memoria 2012_{16} contenga 18_{16}. Dopo l'esecuzione dell'istruzione

LDI

la locazione di memoria 4545_{16} conterrà 18_{16}, la coppia di registri BC conterrà $004E_{16}$, DE conterrà 4546_{16} e HL conterrà 2013_{16}.

LDIR — TRASFERISCE DATI TRA LOCAZIONI DI MEMORIA FINCHE' IL CONTATORE DEI BYTE NON SIA ZERO. INCREMENTA GLI INDIRIZZI DELLA DESTINAZIONE E DELLA SORGENTE

$$\underbrace{\text{LDIR}}_{\text{ED B0}}$$

Questa istruzione è identica a LDI, tranne che essa si ripete finchè la coppia di registri BC non contiene zero. Dopo ogni trasferimento di un dato, si riconosceranno le interruzioni e si eseguiranno due cicli di rinfresco.

Supponiamo di avere i seguenti contenuti nella memoria e nelle coppie di registri:

Registro/Contenuto		Locazione/Contenuto	
HL	2012_{16}	2012_{16}	18_{16}
DE	4545_{16}	2013_{16}	CD_{16}
BC	0003_{16}	2014_{16}	$F0_{16}$

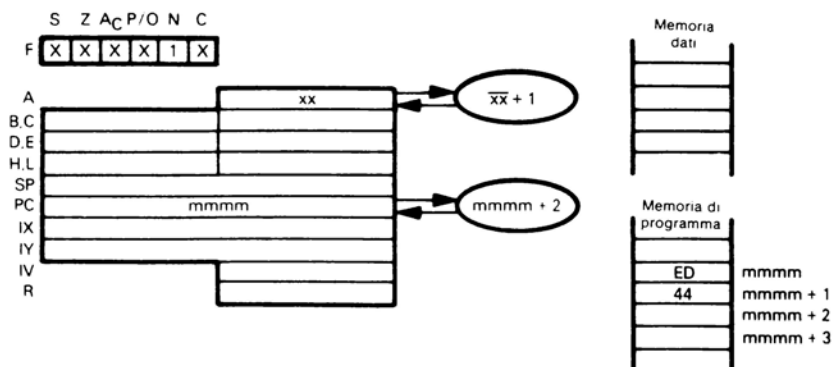
Dopo l'esecuzione di

LDIR

le coppie di registri e la memoria avranno i seguenti contenuti:

Registro/Contenuto		Locazione/Contenuto		Locazione/Contenuto	
HL	2015_{16}	2012_{16}	18_{16}	4545_{16}	18_{16}
DE	4548_{16}	2013_{16}	CD_{16}	4546_{16}	CD_{16}
BC	0000_{16}	2014_{16}	$F0_{16}$	4547_{16}	$F0_{16}$

Questa istruzione è estremamente utile per trasferire blocchi di dati da un'area di memoria ad un'altra.

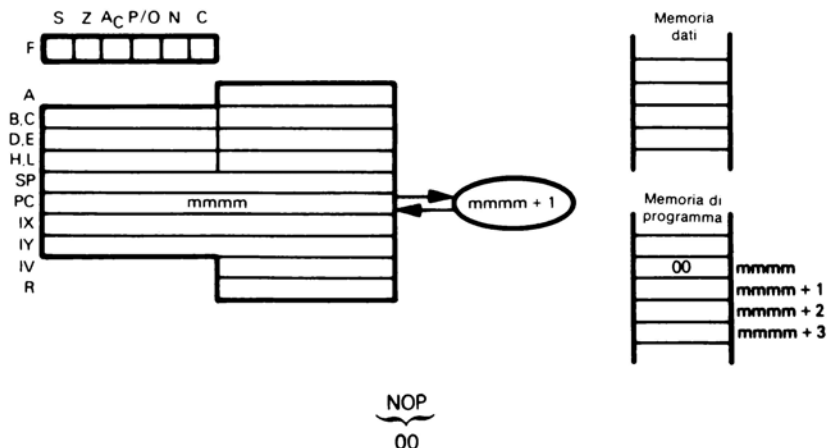
NEG — FA IL COMPLEMENTO A DUE DEL CONTENUTO DELL'ACCUMULATORE

Fa il complemento a due del contenuto dell'Accumulatore. E' lo stesso che sottrarre il contenuto dell'Accumulatore da zero. Il risultato è il complemento a due. 80H rimarrà immutato.

Supponiamo che xx=$5A_{16}$. Dopo l'esecuzione dell'istruzione

NEG

l'Accumulatore conterrà $A6_{16}$.

5A	=	0 1 0 1	1 0 1 0
Complemento a due	=	1 0 1 0	0 1 1 0

NOP — NESSUNA OPERAZIONE

Questa è un'istruzione ad un solo byte che non effettua nessuna operazione, tranne che incrementare il Contatore di Programma e continuare il rinfresco della memoria. Questa istruzione è presente per parecchi motivi:

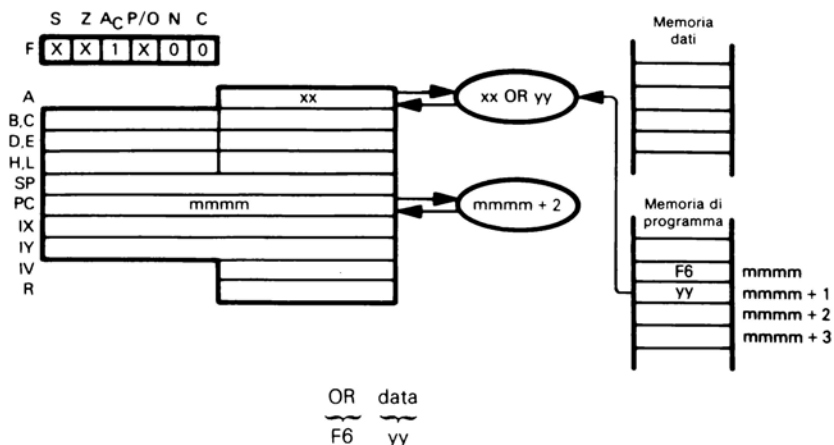
1) Un errore di programma che riporta un codice oggetto da una memoria non esistente riporta 00. E' una buona idea per assicurarsi che l'errore di programma più comune non farà niente.

2) L'istruzione NOP permette di dare un'etichetta ad un byte di codice oggetto: HERE NOP

3) Per tempi di ritardo a regolazione fine. Ogni istruzione NOP somma il ritardo quattro cicli di clock.

NOP non è un'istruzione molto utile o usata frequentemente.

OR data — OR IMMEDIATO CON L'ACCUMULATORE

OR data
F6 yy

Fa l'OR dell'Accumulatore col contenuto del secondo byte del codice oggetto della istruzione

Supponiamo che xx=$3A_{16}$. Dopo l'esecuzione dell'istruzione

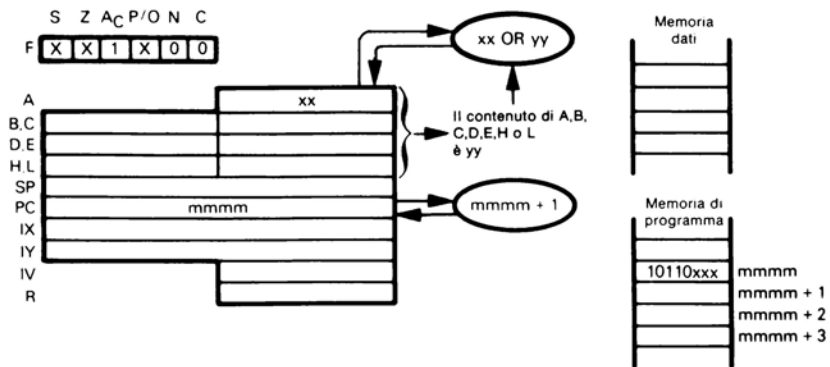
OR 7CH

l'Accumulatore conterrà $7E_{16}$.

Questa è un'istruzione logica di programma; essa è spesso usata per porre dei bit "on". Per esempio l'istruzione

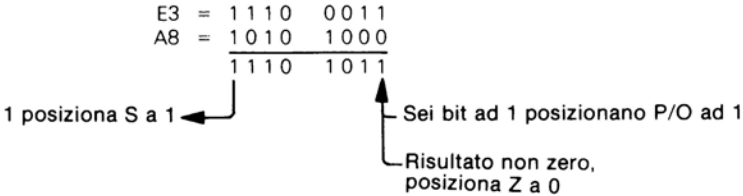
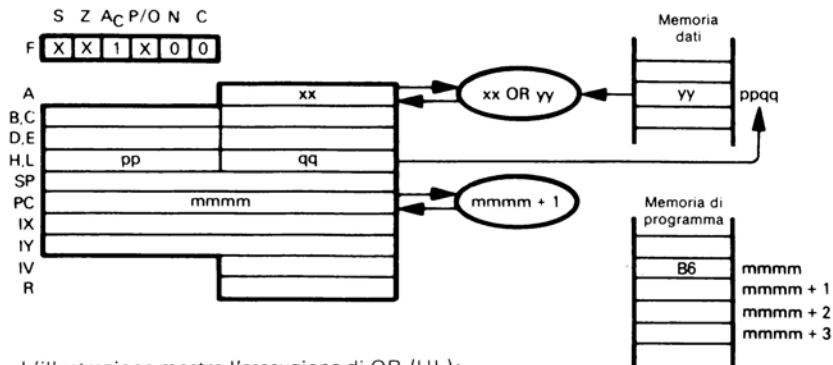
OR 80H

Posizionerà incondizionatamente ad 1 i bit di ordine maggiore dell'Accumulatore.

OR reg — OR DEL REGISTRO CON L'ACCUMULATORE

Fa l'OR logico del contenuto dell'Accumulatore col contenuto del Registro A, B, C, D, E, H o L. Immagazzina il risultato nell'Accumulatore.

Supponiamo che xx=$E3_{16}$ e che il Registro E contenga $A8_{16}$. Dopo l'esecuzione della istruzione

OR E

l'Accumulatore conterrà EB_{16}.

OR (HL) — OR DELLA MEMORIA CON L'ACCUMULATORE
OR (IX + disp)
OR (IY + disp)

L'illustrazione mostra l'esecuzione di OR (HL):

$$\underbrace{\text{OR (HL)}}_{\text{B6}}$$

Fa l'OR del contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) con l'Accumulatore.

Supponiamo che xx=$E3_{16}$, ppqq=4000_{16} e che la locazione di memoria 4000_{16} contenga $A8_{16}$. Dopo l'esecuzione dell'istruzione

OR (HL)

l'Accumulatore conterrà EB_{16}.

```
E3 = 1110  0011
A8 = 1010  1000
     ----------
     1110  1011
```

1 posiziona S a 1

Sei bit ad 1 posizionano P/O ad 1

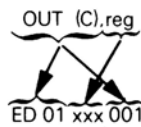
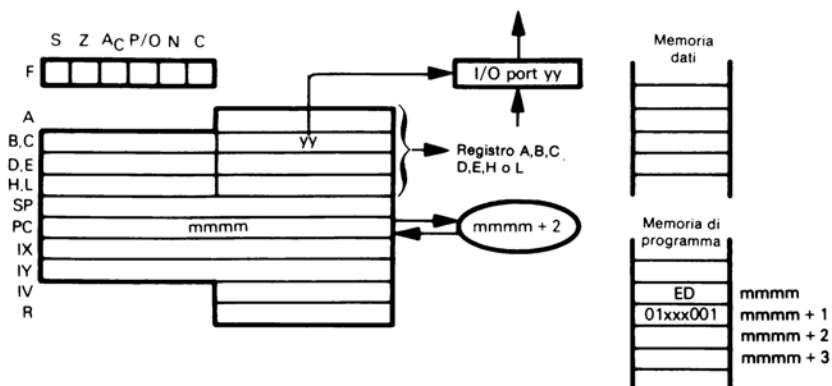
Risultato non zero, posiziona Z a 0

$$\underbrace{\text{OR (IX+}}_{\text{DD B6}}\underbrace{\text{disp)}}_{\text{d}}$$

Fa l'OR del contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del valore d del dislocamento) con l'Accumulatore.

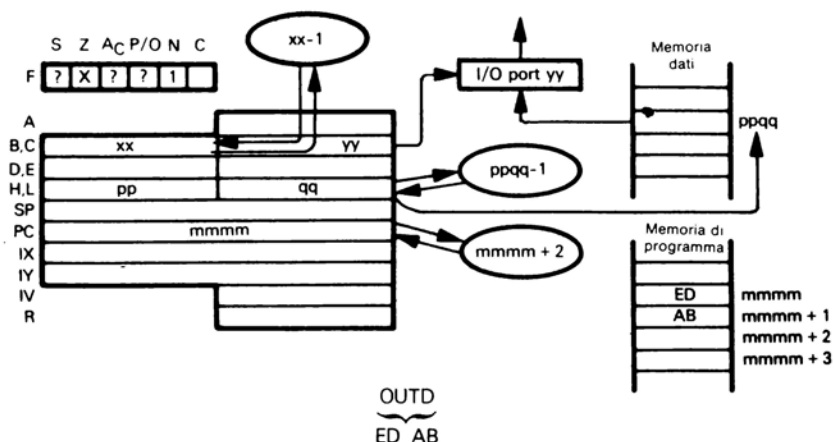
$$\underbrace{\text{OR (IY+}}_{\text{FD B6}}\underbrace{\text{disp)}}_{\text{d}}$$

Questa istruzione è identica a OR (IX + disp), tranne che essa usa il reigstro IY invece del registro IX.

OUT (C),reg — USCITA DA UN REGISTRO

Supponiamo che yy=$1F_{16}$ e che il contenuto di H sia AA_{16}. Dopo l'esecuzione di

OUT (C),H

AA_{16} sarà nel buffer della porta di I/O $1F_{16}$.

OUTD — USCITA DALLA MEMORIA, DECREMENTA L'INDIRIZZO

OUTD
ED AB

Uscita dalla locazione di memoria specificata da HL verso la porta di I/O indirizzata dal Registro C. Si decrementano i registri B e HL.

Supponiamo che xx=$0A_{16}$, yy=FF_{16}, ppqq=5000_{16} e che la locazione di memoria 5000_{16} contenga 77_{16}. Dopo l'esecuzione dell'istruzione

OUTD

nel buffer della porta di I/O FF_{16} sarà contenuto 77_{16}. Il registro B conterrà 09_{16} e la coppia di registri HL conterrà $4FFF_{16}$.

OTDR — USCITA DALLA MEMORIA. DECREMENTA L'INDIRIZZO, CONTINUA FINCHE' IL REGISTRO B =0

OTDR
ED BB

OTDR è identica a OUTD, ma si ripete finchè il Registro B non contenga 0.

Supponiamo che il Registro B contenga 03_{16}, che il Registro C contenga FF_{19} e che HL contenga 5000_{16}. Le locazioni di memoria tra $4FFE_{16}$ e 5000_{16} contengano:

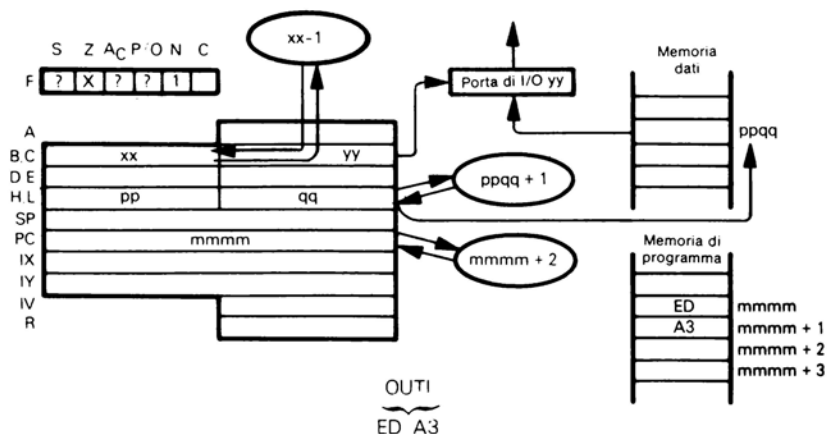
Locazione	Contenuto
$4FFE_{16}$	CA_{16}
$4FFF_{16}$	$1B_{16}$
5000_{16}	$F1_{16}$

Dopo l'esecuzione di

OTDR

la coppia di registri HL conterrà $4FFD_{16}$, il Registro B conterrà zero e nella porta di I/O FF_{16} ci sarà stata scritta la sequenza $F1_{16}$, $1B_{16}$, CA_{16}.

Questa istruzione è molto utile per trasferire blocchi di dati dalla memoria a dispositivi di uscita.

OUTI — USCITA DALLA MEMORIA. INCREMENTO DELL'INDIRIZZO.

OUTI
ED A3

Uscita dalla locazione di memoria specificata da HL verso la porta di I/O indirizzata dal Registro C. Si decrementa il Registro B e si incrmenta la coppia di registri HL.

Supponiamo che xx=$0A_{16}$, yy=FF_{16}, ppqq=5000_{16} e che la locazione di memoria 5000_{16} contenga 77_{16}. Dopo l'esecuzione dell'istruzione

OUTI

il buffer della porta di I/O FF_{16} conterrà 77_{16}. Il registro B conterrà 09_{16} e la coppia di registri HL conterrà 5001_{16}.

OTIR — USCITA DALLA MEMORIA. INCREMENTA L'INDIRIZZO, CONTINUA FINCHE' IL REGISTRO B = 0

OTIR
ED B3

OTIR è identica a OUTI, tranne che essa si ripete finchè il Registro B non contiene 0.

Supponiamo che il Registro B contenga 04_{16}, che il Registro C contenga FF_{16} e che HL contenga 5000_{16}. Le locazioni di memoria da 5000_{16} a 5003_{16} contengono:

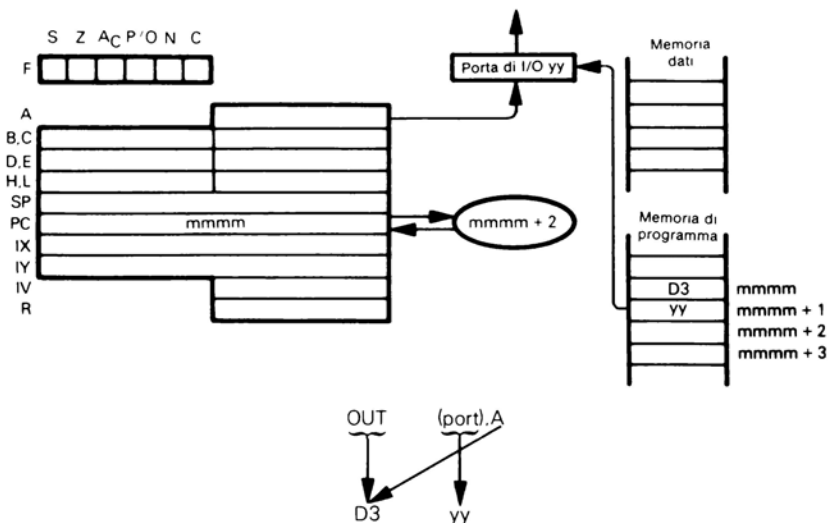
Locazione/Contenuto	
5000_{16}	CA_{16}
5001_{16}	$1B_{16}$
5002_{16}	$B1_{16}$
5003_{16}	AD_{16}

Dopo l'esecuzione di

OTIR

la coppia di registri HL conterrà 5004_{16}, il Registro B conterrà zero e nella porta di I/O FF_{16} sarà stata scritta la sequenza CA_{16}, $1B_{16}$, $B1_{16}$ e AD_{16}.

Questa istruzione è molto utile per trasferire blocchi di dati dalla memoria ad un dispositivo di uscita.

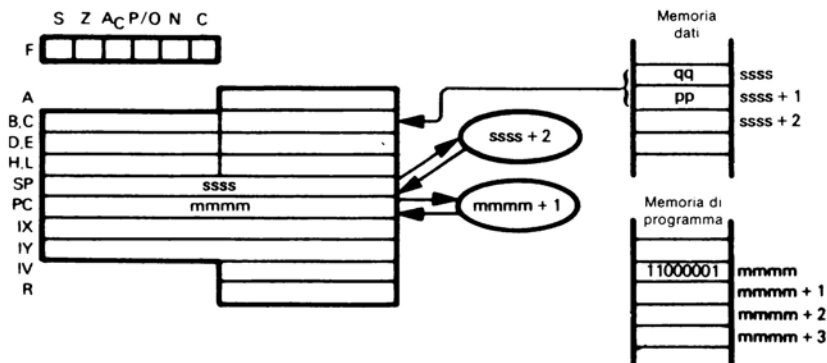
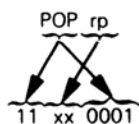
OUT (port),A — USCITA DALL'ACCUMULATORE

Fa uscire il contenuto dell'Accumulatore verso la porta di I/O identificata dal secondo byte del codice oggetto dell'istruzione OUT.

Supponiamo che l'Accumulatore contenga 36_{16}. Dopo l'esecuzione dell'istruzione

OUT (1AH),A

il buffer della porta di I/O $1A_{16}$ conterrà 36_{16}.

L'istruzione OUT non influenza nessuno stato. L'uso dell'istruzione OUT è molto dipendente dall'hardware. Gli indirizzi validi per la porta di I/O sono determinati dal modo con cui si è implementata la logica di I/O. E' inoltre possibile progettare un sistema a microcalcolatore che acceda alla logica esterna usando istruzioni di riferimento alla memoria con specifici indirizzi di memoria. Le istruzioni OUT sono usate frequentemente, in modo speciale per controllare la logica del microcalcolatore esterno alla CPU.

POP rp — LETTURA DALLA SOMMITA' DELLO STACK
POP IX
POP IY

L'illustrazione mostra l'esecuzione di POP BC:

POP rp

11 xx 0001

00 per rp è la coppia di registri BC
01 per rp è la coppia di registri DE
10 per rp è la coppia di registri HL
11 per rp è la coppia di registri A e F

Estrae (POP) i due byte in cima allo stack e li mette nella coppia di registri indicata.

Supponiamo che qq=01_{16} e che pp=$2A_{16}$. L'esecuzione di

POP HL

carica 01_{16} nel registro L e $2A_{16}$ nel registro H. L'esecuzione dell'istruzione

POP AF

carica 01_{16} nei flag degli stati e $2A_{16}$ nell'Accumulatore. In tale modo, lo stato di Carry sarà posizionato ad 1 e gli altri stati saranno azzerati.

POP IX

DD E1

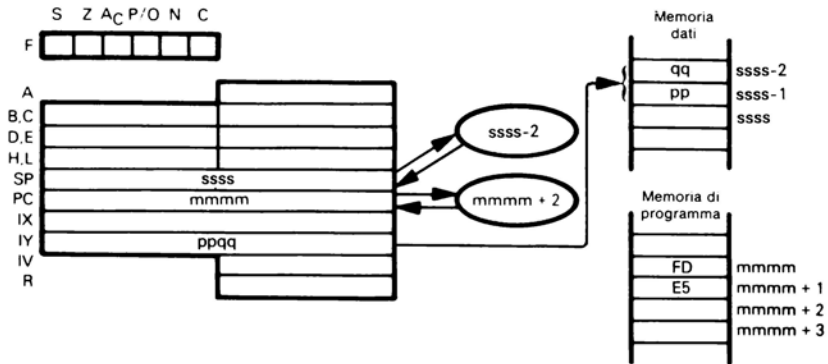
Estrae (POP) i due byte in cima allo stack e li mette nel registro IX.

POP IY

FD E1

Estrae (POP) i due byte in cima allo stack e li mette nel registro IY.

L'istruzione POP è grandemente usata per ripristinare il contenuto dei registri e degli stati che sono stati salvati nello stack; per esempio, durante il servizio di una interruzione.

PUSH rp — SCRITTURA SULLA SOMMITA' DELLO STACK
PUSH IX
PUSH IY

L'illustrazione mostra l'esecuzione di PUSH IY:

PUSH IY
FD E5

Spinge (PUSH) il contenuto del registro IY sulla sommità dello stack.

Supponiamo che il registro IY contenga $45FF_{16}$; l'esecuzione dell'istruzione

PUSH IY

carica 45_{16}, poi FF_{16} sulla sommità dello stack.

PUSH IX
DD E5

Spinge (PUSH) il contenuto del registro IX sulla sommità dello stack.

00 per rp è la coppia di registri BC
01 per rp è la coppia di registri DE
10 per rp è la coppia di registri HL
11 per rp è la coppia di registri A e F

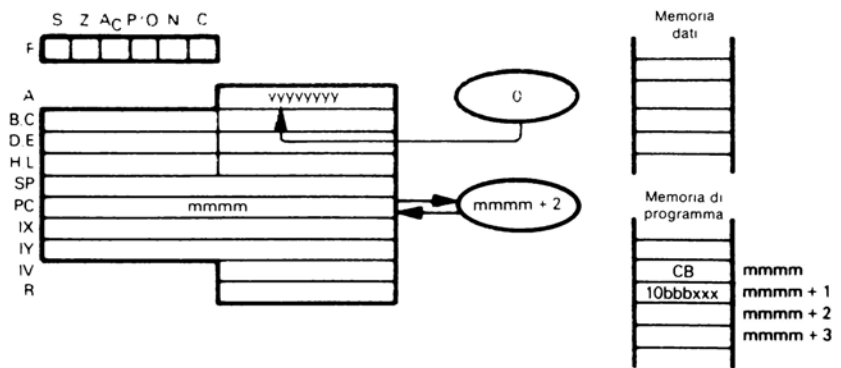
Spinge (PUSH) il contenuto della coppia di registri indicata sulla sommità dello stack.

L'esecuzione dell'istruzione

PUSH AF

carica l'Accumulatore e poi i flag degli stati sulla sommità dello stack.

L'istruzione PUSH è grandemente usata per salvare il contenuto dei registri e degli stati; per esempio, prima di servire un'interruzione.

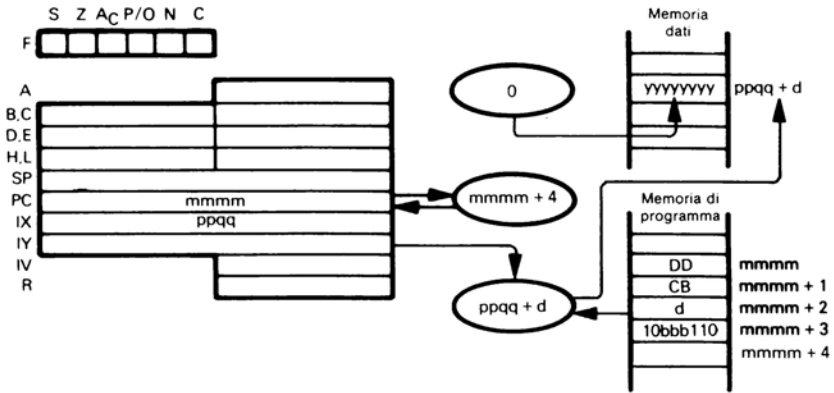
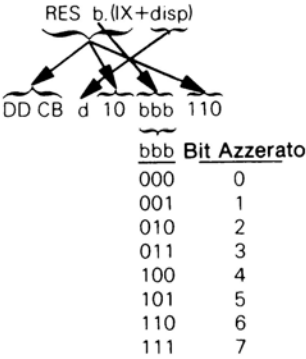
RES b,reg — AZZERA IL BIT DEL REGISTRO INDICATO

Bit	bbb	xxx	Registro
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Azzera il bit indicato nel registro specificato.

Dopo l'esecuzione dell'istruzione

RES 6,H

sarà azzerato il bit 6 nel Registro H. (Il bit 0 è il bit meno significativo).

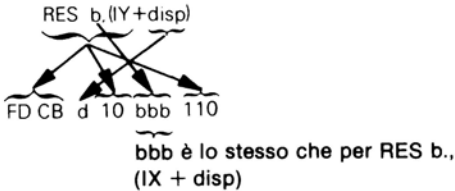
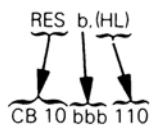
RES b,(HL) — AZZERA IL BIT b DELLA POSIZIONE
RES b,(IX +disp) DI MEMORIA INDICATA
RES b,(IY +disp)

L'illustrazione mostra l'esecuzione di RES b,(IX + disp). Il bit 0 è il bit meno significativo.

bbb	Bit Azzerato
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Azzera il bit indicato nella locazione di memoria indicata dalla somma del Registro Indice IX e d.

Supponiamo che IX contenga 4110_{16}. Dopo l'esecuzione dell'istruzione

RES 0,(IX + 7)

il bit 0 della locazione di memoria 4117_{16} sarà 0.

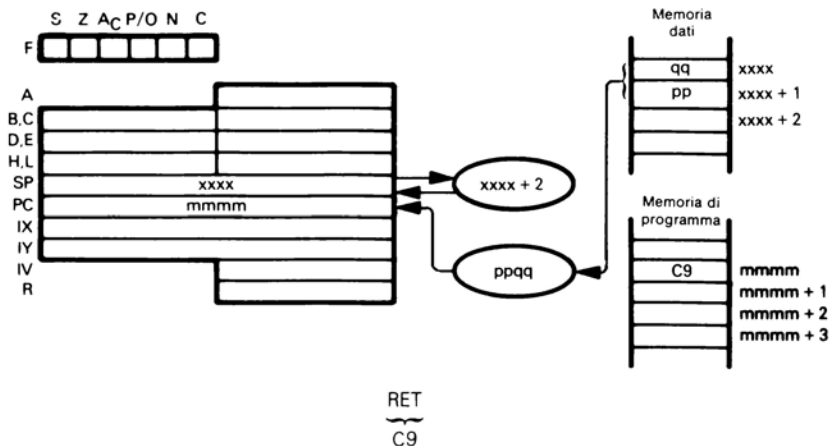
bbb è lo stesso che per RES b., (IX + disp)

Questa istruzione è identica a RES b,(IX + disp), tranne che essa usa il registro IY invece del registro IX.

bbb è lo stesso che per RES b, (IX + disp)

Azzera il bit indicato nella locazione di memoria indicata da HL.

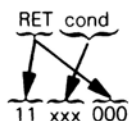
Supponiamo che HL contenga 4444_{16}. Dopo l'esecuzione di

RES 7,(HL)

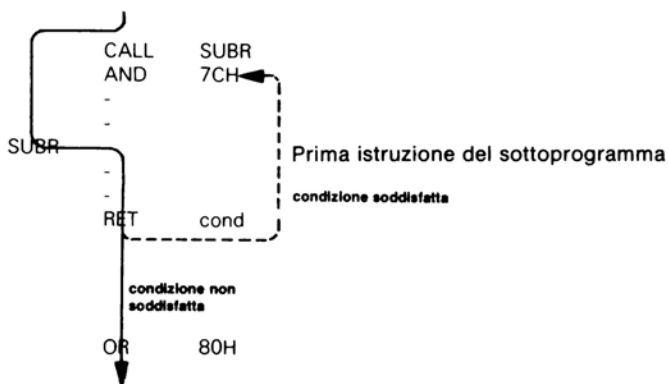
il bit 7 della locazione di memoria 4444_{16} sarà 0.

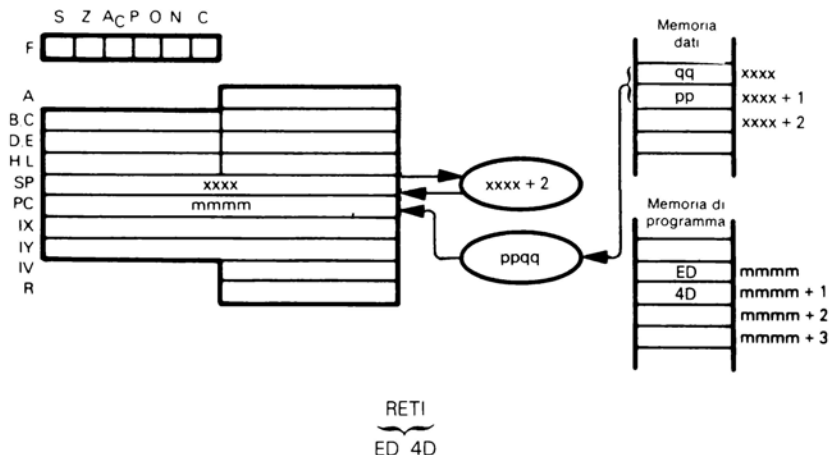
RET — RITORNO DA SOTTOPROGRAMMA

Sposta il contenuto dei due byte in cima allo stack nel Contatore di Programma; questi due byte forniscono l'indirizzo dell'istruzione che si deve eseguire successivamente. Il contenuto precedente del Contatore di Programma è perduto. Incrementa lo Stack Pointer di 2, per indirizzare nuovamente la cima dello stack.

Ogni sottoprogramma deve contenere almeno una istruzione di Return (o un Return condizionato); questa è l'ultima istruzione eseguita nel sottoprogramma e provoca il ritorno all'esecuzione del programma chiamante.

RET cond – RITORNO DA SOTTOPROGRAMMA SE LA CONDIZIONE E' SODDISFATTA

		Condizione	Flag Pertinente
000	NZ	Non-Zero	Z
001	Z	Zero	Z
010	NC	Nessun Carry	C
011	C	Carry	C
100	PO	Parità dispari	P/O
101	PE	Parità pari	P/O
110	P	Segno positivo	S
111	M	Segno negativo	S

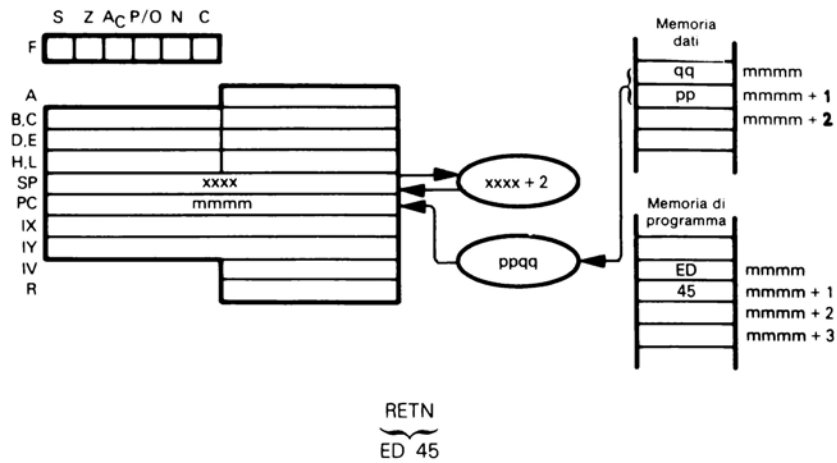
Questa istruzione è identica all'istruzione RET, tranne che non si esegue il ritorno se la condizione non è soddisfatta; altrimenti si eseguirà l'istruzione che segue in sequenza l'istruzione RET cond.

Consideriamo la sequenza di istruzioni:

Dopo l'esecuzione di RET cond, se la condizione è soddisfatta, allora l'esecuzione ritorna all'istruzione AND che segue CALL. Se la condizione non è soddisfatta, sarà eseguita l'istruzione OR, che è la successiva istruzione in sequenza.

RETI — RITORNO DALL'INTERRUZIONE

RETI
ED 4D

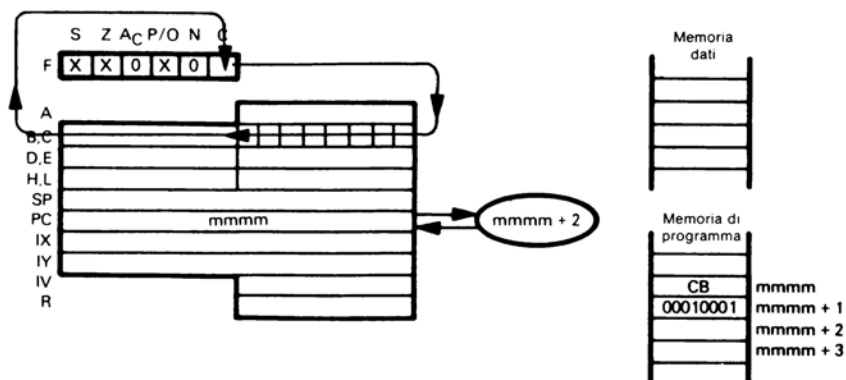
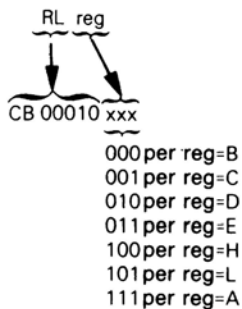
Sposta il contenuto dei due byte in cima allo stack nel Contatore di Programma; questi due byte forniscono l'indirizzo della prossima istruzione da eseguire. Il contenuto precedente del Contatore di Programma è perso. Incrementa lo Stack Pointer di 2 e indirizza nuovamente la cima dello stack.

Questa istruzione è usata alla fine del programma di servizio dell'interruzione e, oltre che per ridare il controllo al programma interrotto, è usata per segnalare ad un dispositivo di I/O che il programma d'interruzione è stato portato a termine. Il dispositivo di I/O deve fornire la logica necessaria per discernere il codice operativo dell'istruzione: si faccia riferimento al Capitolo 7 di An Introduction to Microcomputers: Volume II per una descrizione di come funziona l'istruzione RETI con i dispositivi della famiglia dello Z80.

RETN — RITORNO DA UNA INTERRUZIONE NON MASCHERABILE

RETN
ED 45

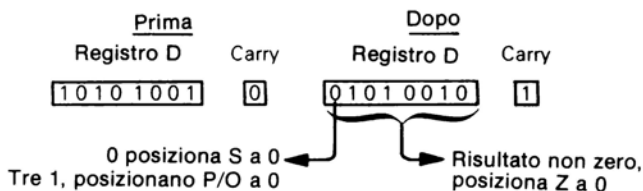
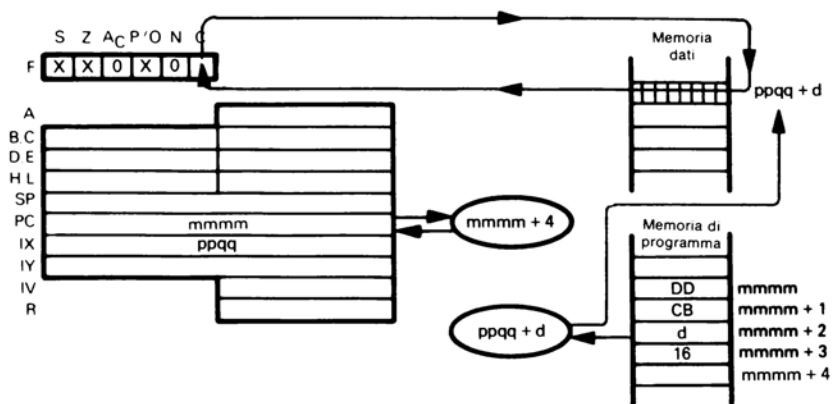
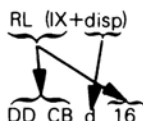
Sposta il contenuto dei due byte in cima allo stack nel Contatore di Programma; questi due byte forniscono l'indirizzo della prossima istruzione da eseguire. Il contenuto precedente del Contatore di Programma è perduto. Incrementa di 2 lo Stack Pointer per indirizzare nuovamente la cima dello stack. Ripristina la logica di abilitazione dell'interruzione per stabilire se essa avesse priorità nell'eventualità di interruzioni non mascherabili.

Questa istruzione è usata alla fine di un programma di servizio di una interruzione non mascherabile e fa sì che l'esecuzione ritorni al programma interrotto.

RL reg — RUOTA IL CONTENUTO DEL REGISTRO A SINISTRA CON CARRY

L'illustrazione mostra l'esecuzione di RL C:

Ruota il contenuto del registro specificato a sinistra di un bit con Carry.

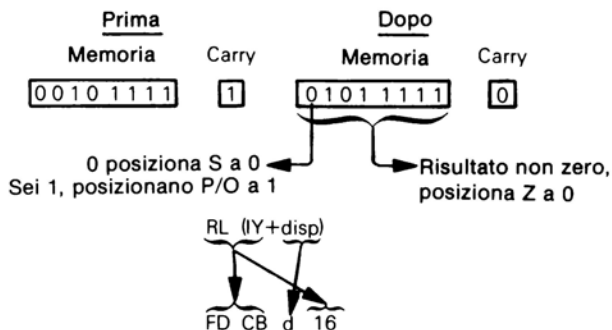
Supponiamo che D contenga $A9_{16}$ e che Carry=0. Dopo l'esecuzione dell'istruzione

RL D

D conterrà 52_{16} e il Carry sarà uguale a 1:

RL (HL) — RUOTA IL CONTENUTO DELLA LOCAZIONE
RL (XI + disp) DI MEMORIA A SINISTRA CON CARRY
RL (IY + disp)

L'illustrazione mostra l'esecuzione di RL (IX + disp):

Ruota il contenuto della locazione di memoria (specificata dalla somma del contenuto del Registro Indice IX e del valore intero d del dislocamento) a sinistra di un bit con Carry.

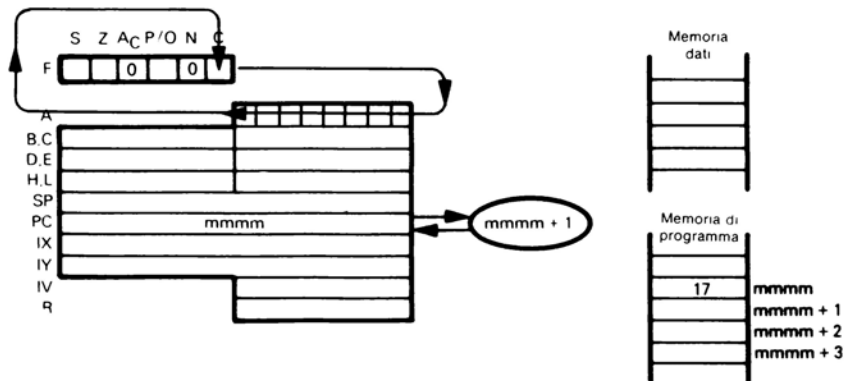
Supponiamo che il registro IX contenga 4000_{16}, che la locazione di memoria 4007_{16} contenga $2F_{16}$ e che il Carry sia posto a 1. Dopo l'esecuzione dell'istruzione

RL (X + 7)

la locazione di memoria 4007_{16} conterrà $5F_{16}$ e il Carry sarà 0:

Questa istruzione è identica a RL (IX + disp), ma usa il registro IY invece del registro IX.

RL (HL)
CB 16

Ruota il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) a sinistra di un bit con Carry.

RLA — RUOTA L'ACCUMULATORE A SINISTRA CON CARRY

RLA
17

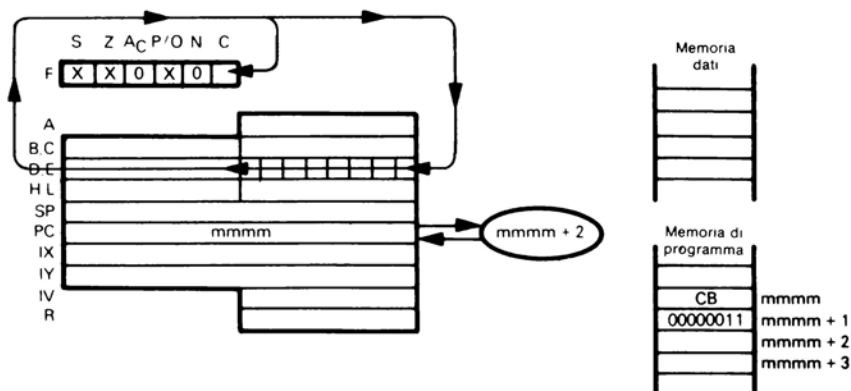
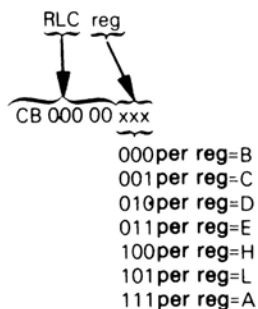
Ruota il contenuto dell'Accumulatore a sinistra di un bit con lo stato di Carry.

Supponiamo che l'Accumulatore contenga $2A_{16}$ e che lo stato di Carry sia posizionato ad 1. Dopo l'esecuzione dell'istruzione

RLA

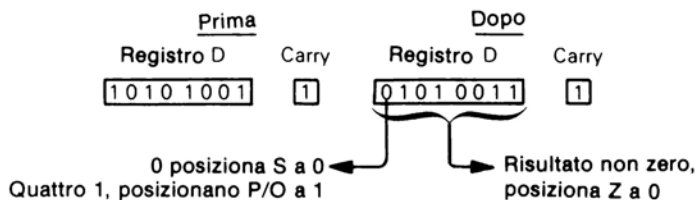
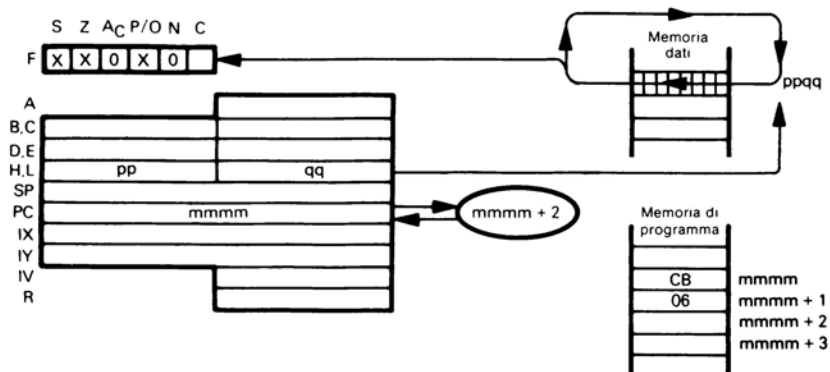
l'Accumulatore conterrà $F5_{16}$ e lo stato di Carry sarà posizionato a 0:

Prima		Dopo	
Accumulatore	Carry	Accumulatore	Carry
0 1 1 1 1 0 1 0	1	1 1 1 1 0 1 0 1	0

RLC reg — RUOTA IL CONTENUTO DEL REGISTRO A SINISTRA CON RICIRCOLO

L'illustrazione mostra l'esecuzione di RLC E:

Ruota il contenuto del registro specificato a sinistra di un bit, ricopiando il bit 7 nel Carry.

Supponiamo che il Registro D contenga $A9_{16}$ e che il Carry sia 1. Dopo l'esecuzione di

RLC D

il Registro D conterrà 53_{16} e il Carry sarà 1:

RLC (HL) — RUOTA IL CONTENUTO DELLA LOCAZIONE
RLC (IX +disp) DI MEMORIA A SINISTRA CON RICIRCOLO
RLC (IY +disp)

L'illustrazione mostra l'esecuzione di RLC (HL):

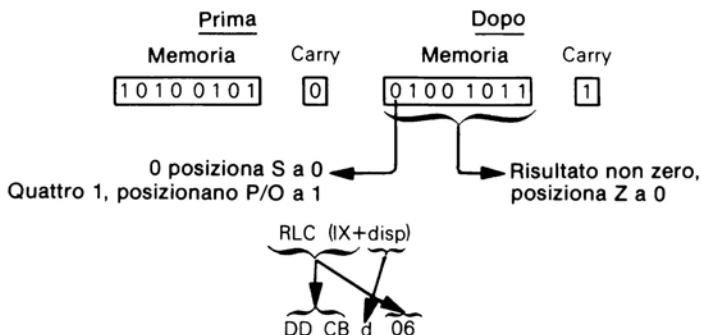
RLC (HL)
CB 06

Ruota il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) a sinistra di un bit, ricopiando il bit 7 nel Carry.

Supponiamo che la coppia di registri HL contenga $54FF_{16}$. La locazione di memoria $54FF_{16}$ contenga $A5_{16}$ e il Carry sia 0. Dopo l'esecuzione di

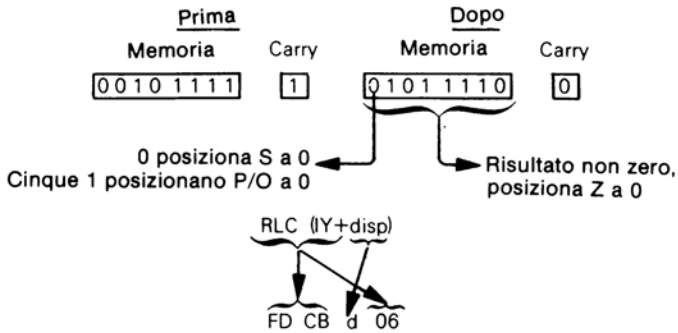
RLC (HL)

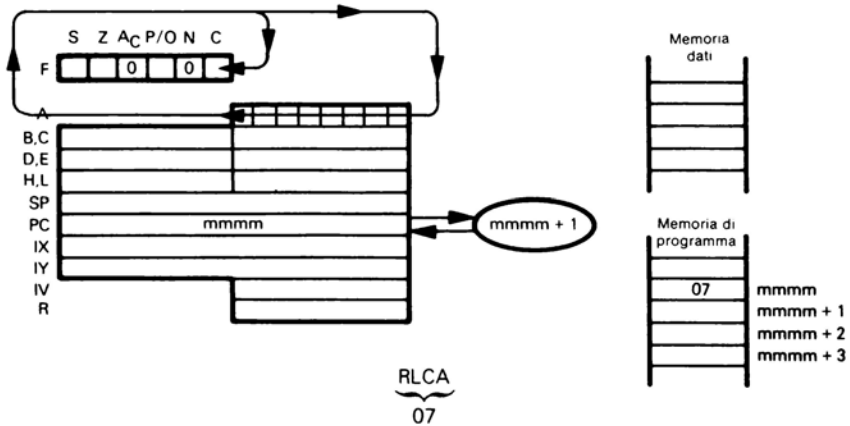
la locazione di memoria $54FF_{16}$ conterrà $4B_{16}$ e il Carry sarà 1:

RLC (IX+disp)
DD CB d 06

Ruota la locazione di memoria (specificata dalla somma del contenuto del registro Indice IX e del valore intero d del dislocamento) a sinistra di un bit, ricopiando il bit 7 nel Carry.

Supponiamo che il registro IX contenga 4000_{16}. Il Carry sia 1 e la locazione di memoria 4007_{16} contenga $2F_{16}$. Dopo l'esecuzione dell'istruzione

RLC (IX + 7)

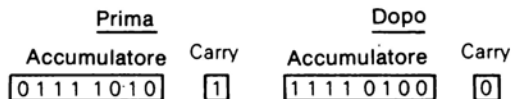
la locazione di memoria 4007_{16} conterrà $5E_{16}$ e il Carry sarà 0:

Questa istruzione è identica a RLC (IX + disp), ma usa il registro IY invece del registro IX.

RLCA — RUOTA L'ACCUMULATORE A SINISTRA CON RICIRCOLO

Ruota il contenuto dell'Accumulatore a sinistra di un bit, ricopiando il bit 7 nel Carry.

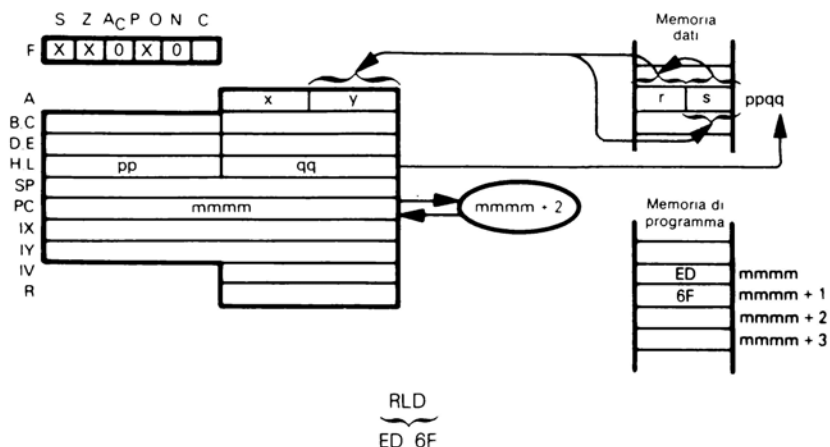
Supponiamo che l'Accumulatore contenga $7A_{16}$ e che lo stato del Carry sia posizionato ad 1. Dopo l'esecuzione dell'istruzione

RLCA

l'Accumulatore conterrà $F4_{16}$ e lo stato del Carry sarà posizionato a 0:

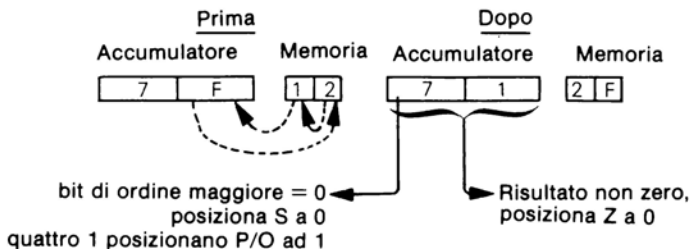
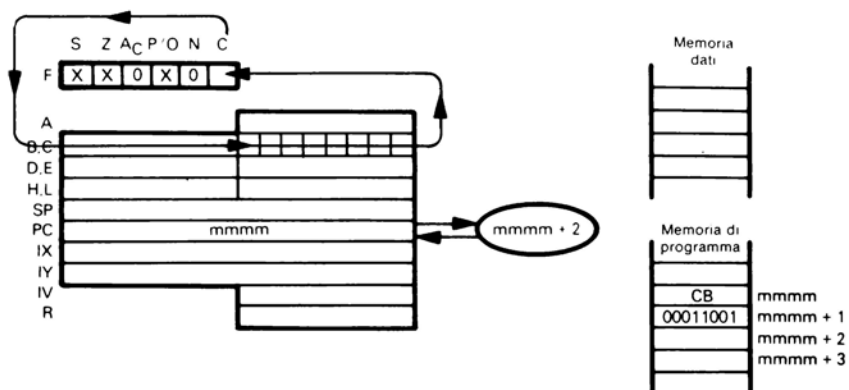
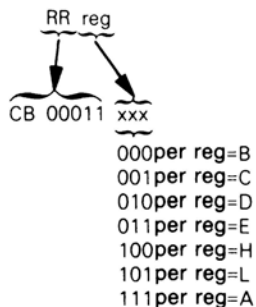
	Prima		Dopo	
	Accumulatore	Carry	Accumulatore	Carry
	0 1 1 1 1 0 1 0	1	1 1 1 1 0 1 0 0	0

RLCA dovrebbe essere usata come istruzione logica.

RLD — RUOTA UN DIGIT BCD A SINISTRA TRA L'ACCUMULATORE E UNA LOCAZIONE DI MEMORIA

RLD
ED 6F

I quattro bit di ordine minore di una locazione di memoria (specificata dal contenuto della coppia di registri HL) sono ricopiati nei quattro bit di ordine maggiore della stessa locazione di memoria. Il contenuto precedente dei quattro bit di ordine maggiore sono ricopiati nei quattro bit di ordine minore dell'Accumulatore. I precedenti quattro bit di ordine minore dell'Accumulatore sono ricopiati nei quattro bit di ordine minore della locazione di memoria specificata.

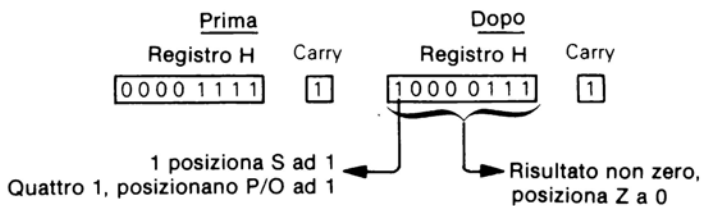
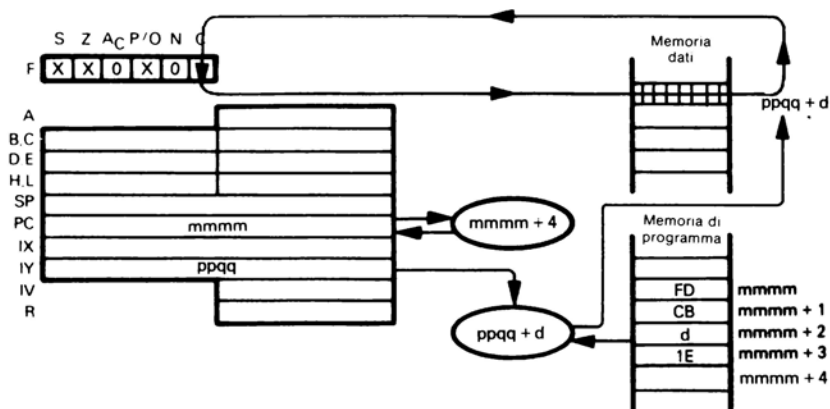
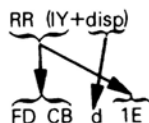
Supponiamo che l'Accumulatore contenga $7F_{16}$, che la coppia di registri HL contenga 4000_{16} e che la locazione di memoria 4000_{16} contenga 12_{16}. Dopo l'esecuzione dell'istruzione

RLD

l'Accumulatore conterrà 71_{16} e la locazione di memoria 4000_{16} conterrà $2F_{16}$:

RR reg — RUOTA IL CONTENUTO DEL REGISTRO A DESTRA CON CARRY

L'illustrazione mostra l'esecuzione di RR C:

Ruota il contenuto del registro specificato a destra di un bit con Carry.

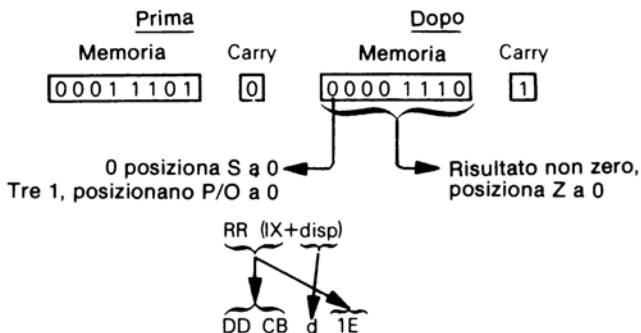
Supponiamo che il Registro H contenga $0F_{16}$ e che il Carry sia posizionato ad 1. Dopo l'esecuzione dell'istruzione

RR H

il Registro H conterrà 87_{16} e il Carry sarà 1:

RR (HL) — RUOTA IL CONTENUTO DELLA LOCAZIONE DI MEMORIA A DESTRA CON CARRY
RR (IX + disp)
RR (IY + disp)

L'illustrazione mostra l'esecuzione di RR (IY + disp):

Ruota il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IY e del valore d del dislocamento) a destra di un bit con Carry.

Supponiamo che il registro IY contenga 4500_{16}; che la locazione di memoria $450F_{16}$ contenga $1D_{16}$ e che il Carry sia posizionato a 0. Dopo l'esecuzione dell'istruzione

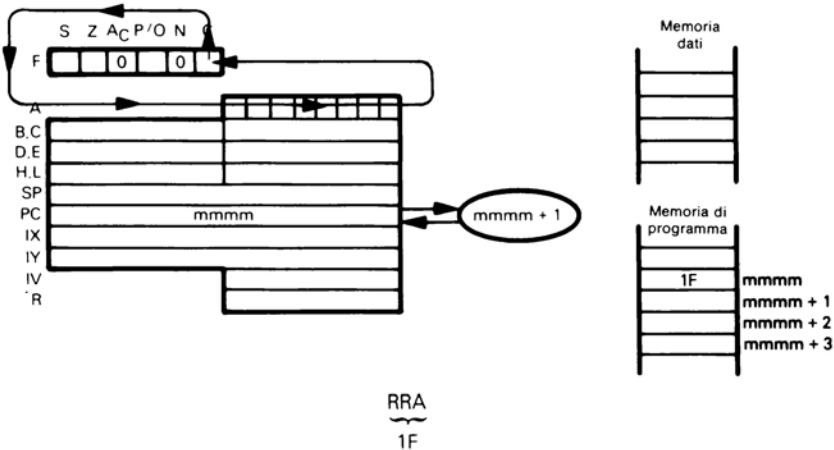
RR (IY + 0FH)

la locazione di memoria $450F_{16}$ conterrà $0E_{16}$ e il Carry sarà 1:

Questa istruzione è identica a RR (IY + disp), ma usa il registro IX invece del registro IY.

RR (HL)
CB 1E

Ruota il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) a destra di un bit con Carry.

RRA — RUOTA L'ACCUMULATORE A DESTRA CON CARRY

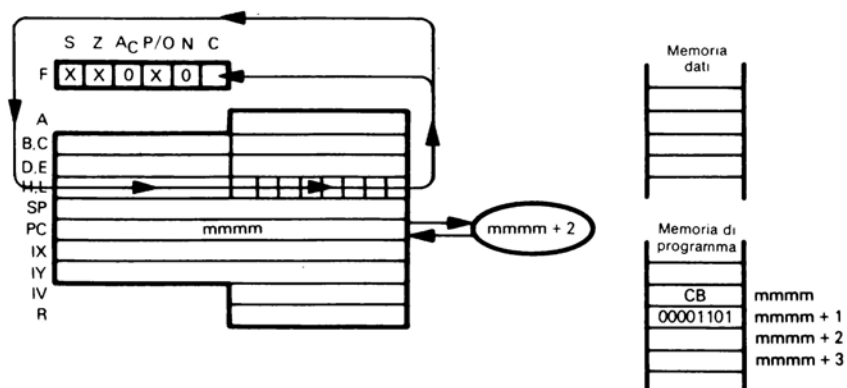
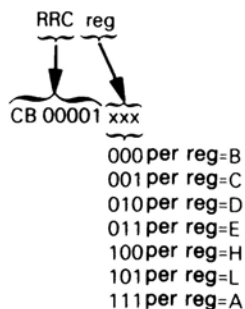
RRA
1F

Ruota il contenuto dell'Accumulatore a destra di un bit con lo stato di Carry.

Supponiamo che l'Accumulatore contenga $7A_{16}$ e che lo stato del Carry sia posizionato ad 1. Dopo l'esecuzione dell'istruzione

RRA

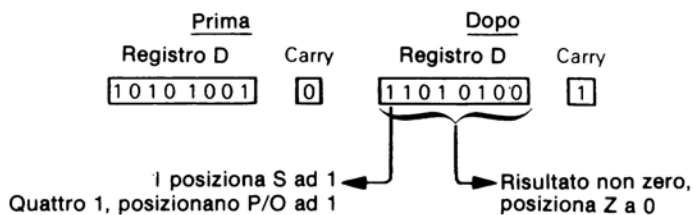
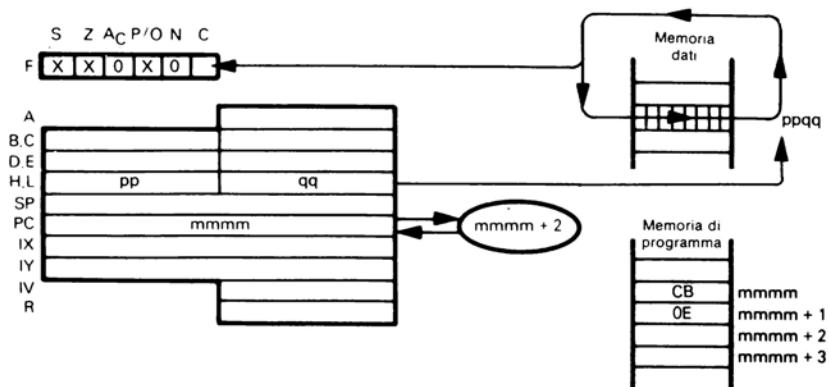
l'Accumulatore conterrà BD_{16} e lo stato del Carry sarà posizionato a 0:

Prima		Dopo	
Accumulatore	Carry	**Accumulatore**	Carry
0111 1010	1	1011 1101	0

RRC reg — RUOTA IL CONTENUTO DEL REGISTRO A DESTRA CON RICIRCOLO

L'illustrazione mostra l'esecuzione di RRC L:

Ruota il contenuto del registro specificato a destra di un bit con ricircolo, ricopiando il bit 0 nello stato del Carry.

Supponiamo che il Registro D contenga $A9_{16}$ e che il Carry sia 0. Dopo l'esecuzione di

RRC D

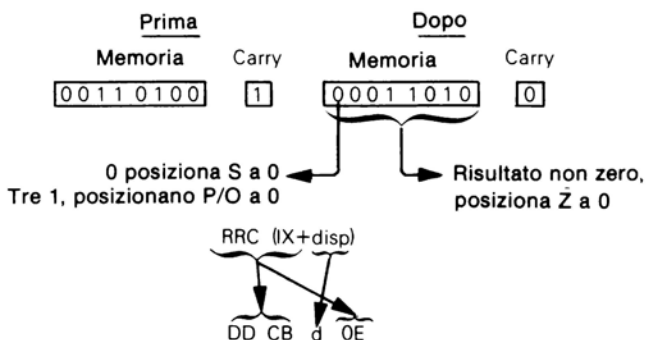
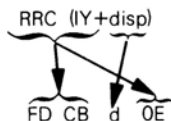
il Registro D conterrà $D4_{16}$ e il Carry sarà 1:

RRC (HL) — RUOTA IL CONTENUTO DELLA LOCAZIONE DI MEMORIA A DESTRA CON RICIRCOLO
RRC (IX +disp)
RRC (IY +disp)

L'illustrazione mostra l'esecuzione di RRC (HL):

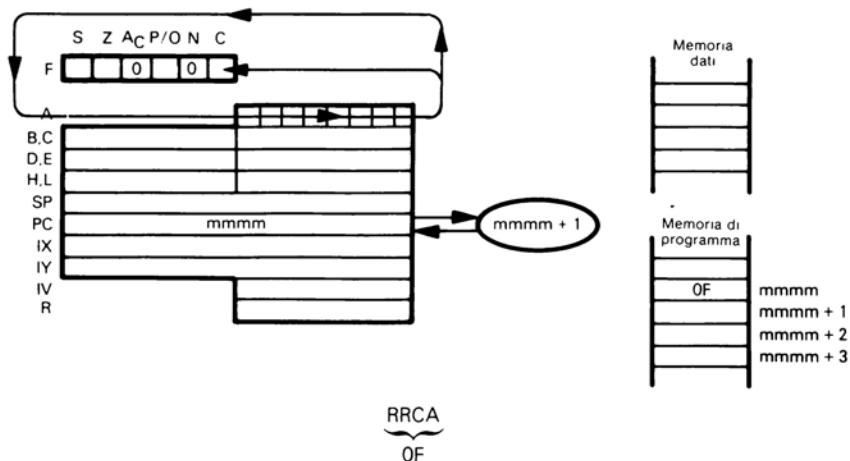
RRC (HL)
CB 0E

Ruota il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) a destra di un bit con ricircolo, ricopiando il bit 0 nello stato del Carry.

Supponiamo che la coppia di registri HL contenga 4500_{16}, che la locazione di memoria 4500_{16} contenga 34_{16} e che il Carry sia posizionato ad 1. Dopo l'esecuzione di

RRC (HL)

la locazione di memoria 4500_{16} conterrà $1A_{16}$ e il Carry sarà 0:

Ruota il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del valore d del dislocamento) a destra di un bit con ricircolo, ricopiando il bit 0 nello stato del Carry.

Questa istruzione è identica all'istruzione RRC (IX + disp), ma usa il registro IY invece del registro IX.

RRCA — RUOTA L'ACCUMULATORE A DESTRA CON RICIRCOLO

RRCA
0F

Ruota il contenuto dell'Accumulatore a destra di un bit con ricircolo, ricopiando il bit 0 nello stato del Carry.

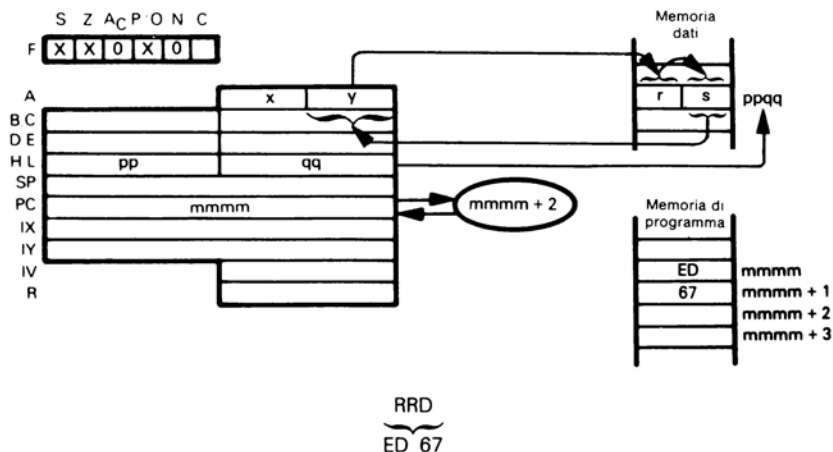
Supponiamo che l'Accumulatore contenga $7A_{16}$ e che lo stato del Carry sia posizionato ad 1. Dopo l'esecuzione dell'istruzione

RRCA

l'Accumulatore contiene $3D_{16}$ e lo stato del Carry sarà posizionato a 0:

Prima		Dopo	
Accumulatore	Carry	Accumulatore	Carry
0111 1010	1	0011 1101	0

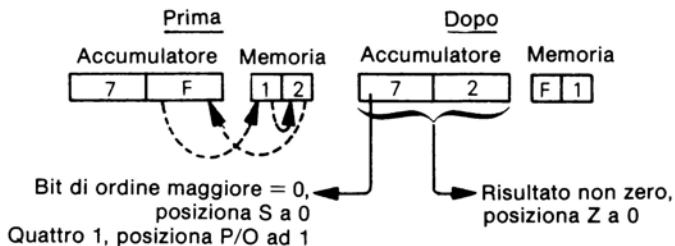
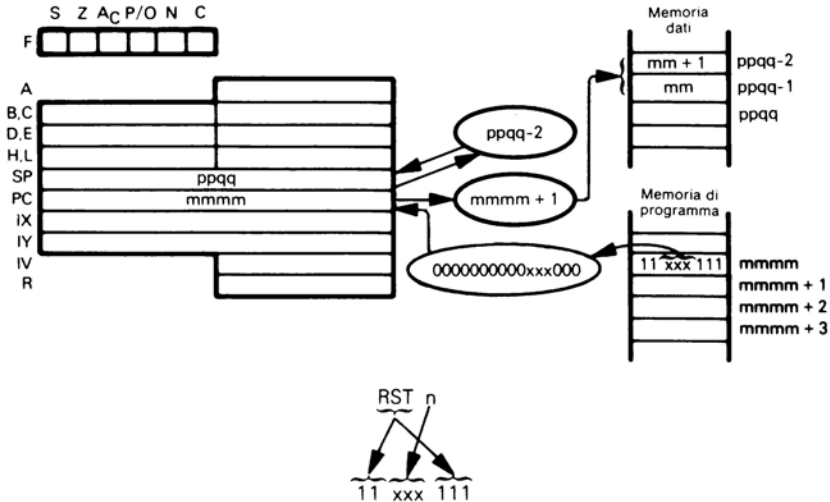
RRCA dovrebbe essere usata come un'istruzione logica.

RRD — RUOTA UN DIGIT BCD A DESTRA TRA L'ACCUMULATORE E LA LOCAZIONE DI MEMORIA

RRD
ED 67

I quattro bit di ordine maggiore della locazione di memoria (specificata dal contenuto della coppia di registri HL) sono ricopiati nei quattro bit di ordine minore della stessa locazione di memoria. Il contenuto precedente dei quattro bit di peso minore è ricopiato nei quattro bit di peso minore dell'Accumulatore. I precedenti quattro bit di ordine minore dell'Accumulatore sono ricopiati nei quattro bit di ordine maggiore della locazione di memoria specificata.

Supponiamo che l'Accumulatore contenga $7F_{16}$, che la coppia di registri HL contenga 4000_{16} e che la locazione di memoria 4000_{16} contenga 12_{16}. Dopo l'esecuzione della istruzione

RRD

l'Accumulatore conterrà 72_{16} e la locazione di memoria 4000_{16} conterrà $F1_{16}$:

RST n — RIAVVIO (RESTART)

Chiama il sottoprogramma originato all'indirizzo basso di memoria specificato da n.

Quando viene eseguita l'istruzione

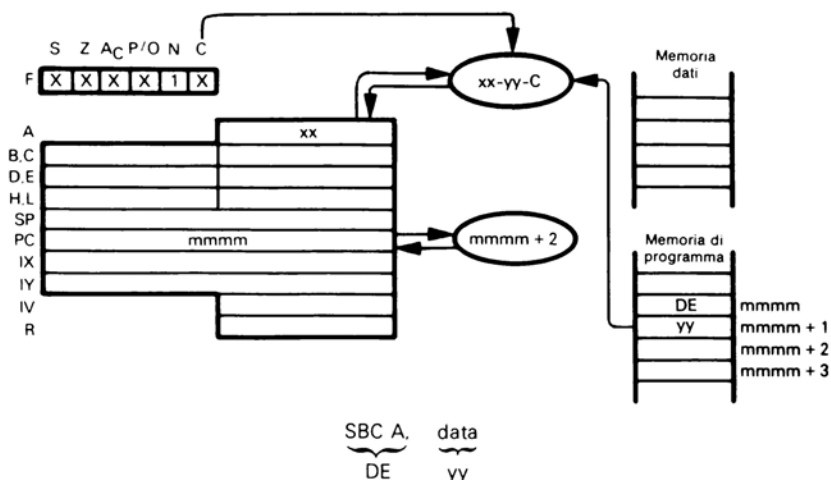
RST 18H

si chiama il sottoprogramma originato alla locazione di memoria 0018_{16}. Il contenuto precedente del Contatore di Programma è spinto in cima allo stack.

Usualmente, l'istruzione RST è usata congiuntamente ad un processo di interruzione, come descritto nel Capitolo 5.

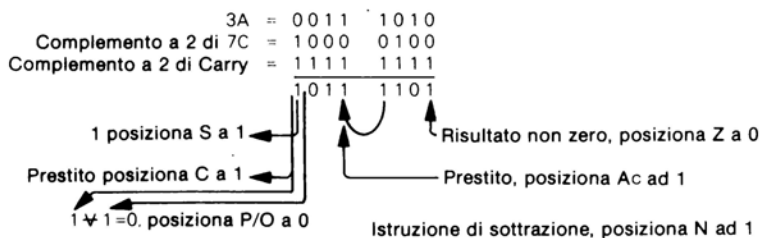
CHIAMATA DI UN SOTTOPROGRAMMA USANDO RST

Se il vostro programma non usa tutti i codici dell'istruzione RST per servire le interruzioni, non trascurate la possibilità di chiamare sottoprogrammi che usano istruzioni RST. Si ponga l'origine di sottoprogrammi frequentemente usati ad indirizzi appropriati di RST; questi sottoprogrammi possono essere chiamati con un'istruzione a singolo byte RST invece che con un'istruzione CALL a tre byte.

SBC A,data — SOTTRAZIONE IMMEDIATA DI UN DATO DALL'ACCUMULATORE CON PRESTITO

Sottrae il contenuto del secondo byte di codice oggetto e lo stato del Carry dall'Accumulatore.

Supponiamo che xx=$3A_{16}$ e che il Carry=1. Dopo l'esecuzione dell'istruzione

SBC A,7CH

l'Accumulatore conterrà BD_{16}.

E' da notare che il carry risultante è complementato.

SBC A,reg — SOTTRAE IL REGISTRO CON PRESTITO DALL'ACCUMULATORE

S Z AC P/O N C
F X X X X 1 X

xx-yy-C

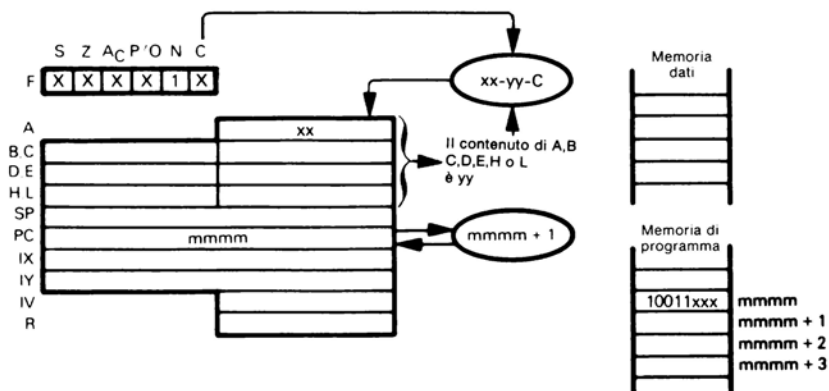
A xx
B C
D E
H L
SP
PC mmmm
IX
IY
IV
R

Il contenuto di A,B C,D,E,H o L è yy

mmmm + 1

Memoria dati

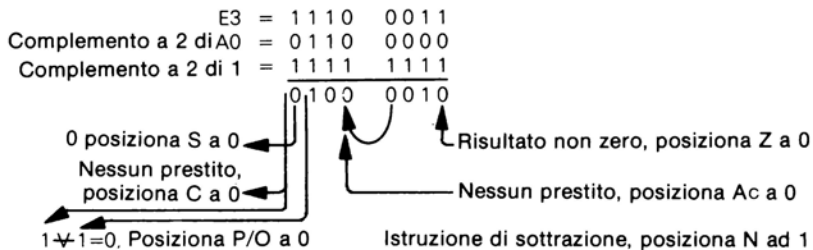
Memoria di programma

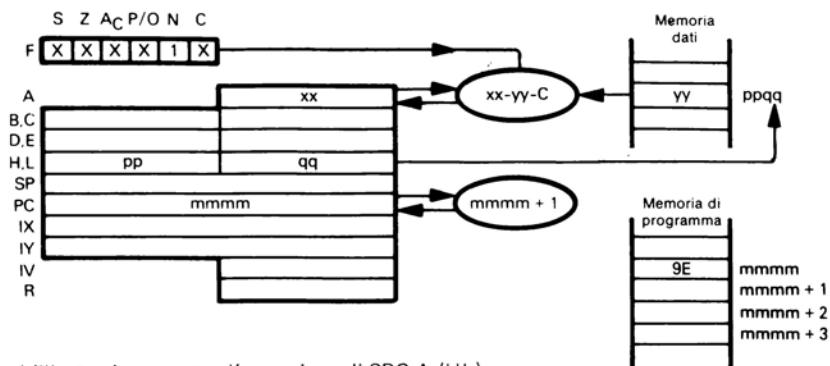
10011xxx mmmm
mmmm + 1
mmmm + 2
mmmm + 3

```
SBC A,    reg
10011     xxx
          000 per reg=B
          001 per reg=C
          010 per reg=D
          011 per reg=E
          100 per reg=H
          101 per reg=L
          111 per reg=A
```

Sottrae il contenuto del registro specificato e lo stato del Carry dall'Accumulatore.

Supponiamo che xx=$E3_{16}$, che il Registro E contenga $A0_{16}$ e che il Carry=1. Dopo l'esecuzione dell'istruzione

SBC A,E

l'Accumulatore conterrà 42_{16}.

E' da notare che il carry risultante è complementato.

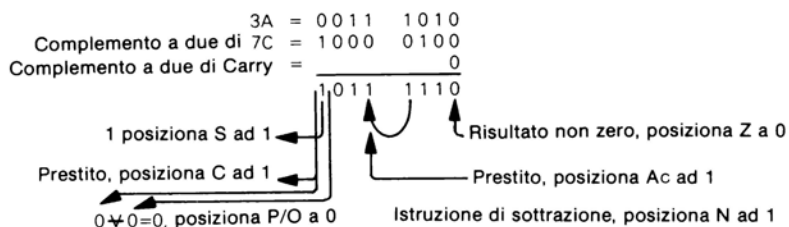
SBC A,(HL) — SOTTRAE LA MEMORIA ED IL CARRY
SBC A,(IX +disp) DALL'ACCUMULATORE
SBC A,(IY +disp)

L'illustrazione mostra l'esecuzione di SBC A,(HL):

SBC A,(HL)
9E

Sottrae il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) e il Carry dall'Accumulatore.

Supponiamo che il Carry=0, ppqq=4000_{16}, xx=$3A_{16}$ e che la locazione di memoria 4000_{16} contenga $7C_{16}$. Dopo l'esecuzione dell'istruzione

SBC A,(HL)

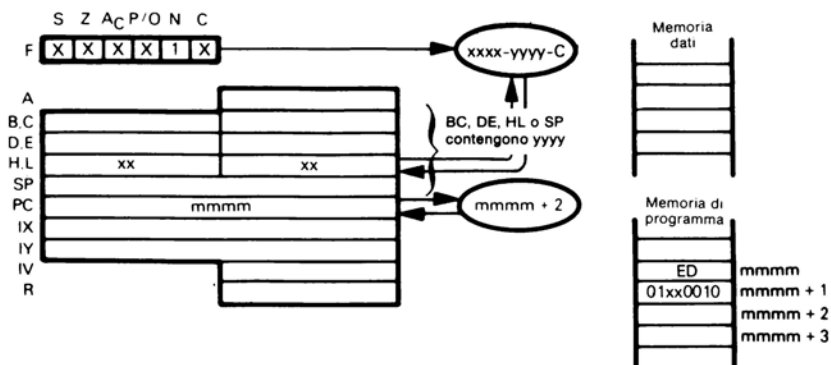
l'Accumulatore conterrà BE_{16}.

E' da notare che il carry risultante è complementato.

SBC A,(IX+disp)
DD 9E d

Sottrae il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del valore d del dislocamento) e il Carry dall'Accumulatore.

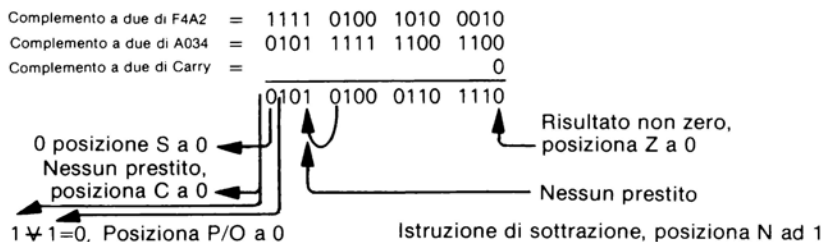
SBC A,(IY+disp)
FD 9E d

Questa istruzione è identica a SBC A,(XI + disp), tranne che essa usa il registro IY invece del registro IX.

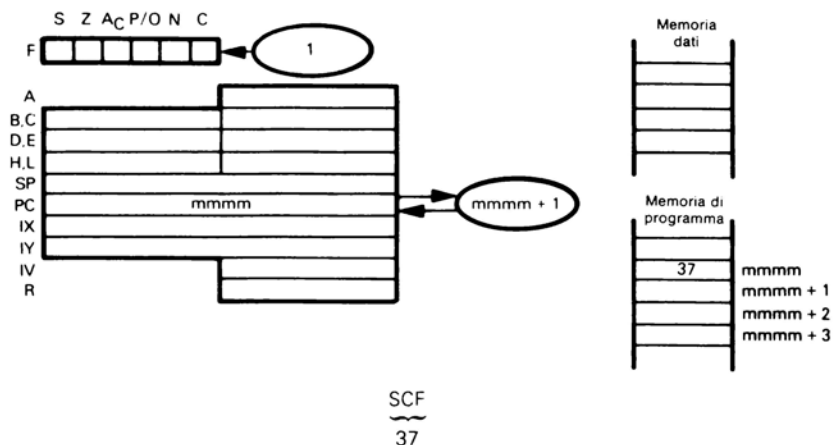
SBC HL,rp — SOTTRAE LA COPPIA DI REGISTRI CON CARRY DA H ED L

Sottrae il contenuto della coppia di registri indicata e lo stato del Carry dalla coppia di registri HL.

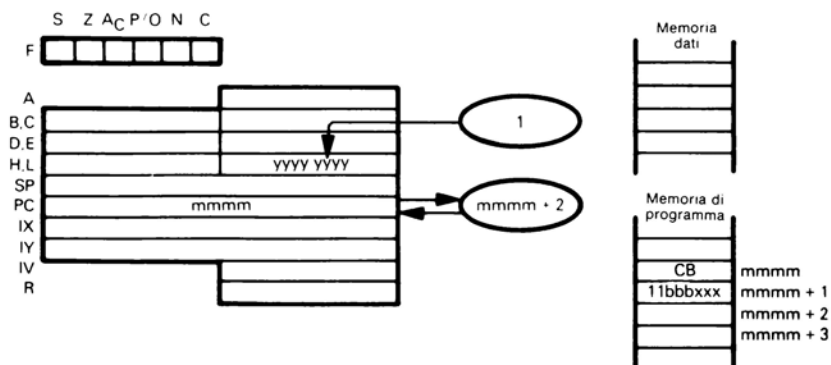
Supponiamo che HL contenga $F4A2_{16}$, che BC contenga $A034_{16}$ e che Carry=0. Dopo l'esecuzione dell'istruzione

SBC HL,BC

la coppia di registri HL conterrà $546E_{16}$:

E' da notare che il carry risultante è complementato.

SCF — POSIZIONA IL FLAG DI CARRY

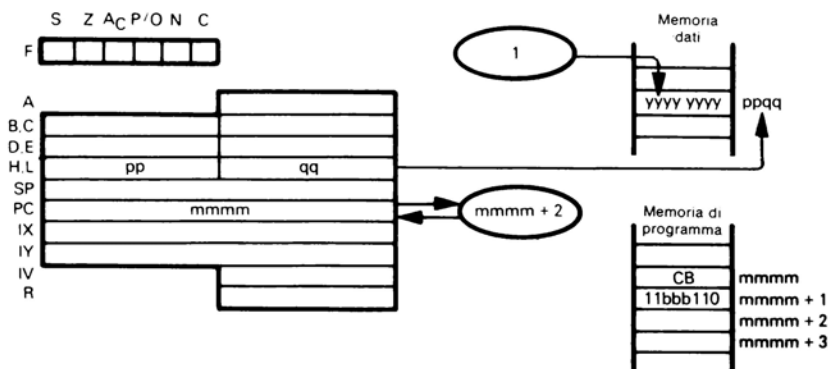
SCF
37

Quando viene eseguita l'istruzione SCF, si posiziona a 1 lo stato del Carry, senza riguardo al suo valore precedente. Non si influenza il contenuto di nessun altro stato o registro.

SET b,reg — POSIZIONA IL BIT INDICATO DEL REGISTRO

Bit	bbb	xxx	Registro
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Il bit indicato da SET nel registro specificato viene posizionato ad 1. Dopo l'esecuzione dell'istruzione

SET 2,L

Il bit 2 del Registro L sarà posiziona ad 1. (Il bit 0 è il bit meno significativo).

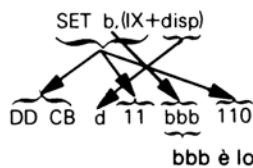
SET b,(HL) — POSIZIONA IL BIT b DELLA POSIZIONE
SET b,(IX + disp) DI MEMORIA INDICATA
SET b,(IY + disp)

L'illustrazione mostra l'esecuzione di SET b(HL). Il bit 0 è il bit meno significativo.

Bit posizionato	bbb
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Posiziona ad 1 il bit indicato nella locazione di memoria indicata da HL.

Supponiamo che HL contenga 4000_{16}. Dopo l'esecuzione dell'istruzione

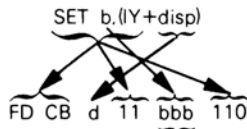
SET 5,(HL)

il bit 5 della posizione di memoria 4000_{16} sarà 1.

bbb è lo stesso di SET b,(HL)

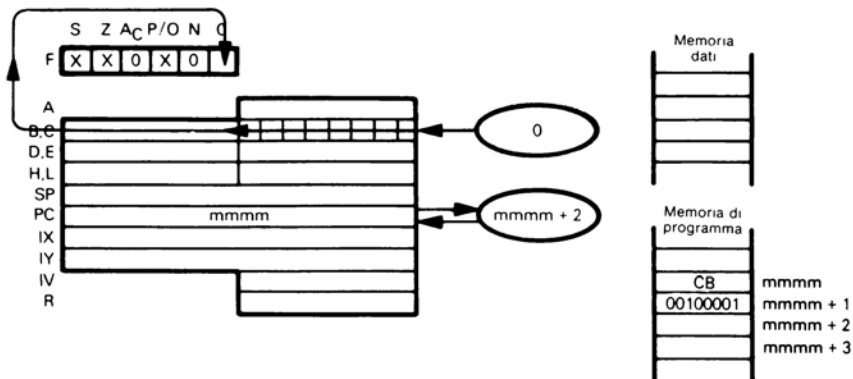
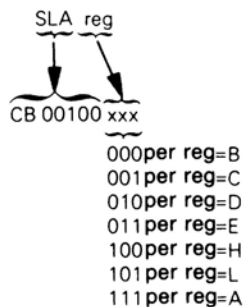
Posiziona ad 1 il bit indicato nella locazione di memoria indicata dalla somma del Registro Indice IX e del dislocamento.

Supponiamo che il Registro Indice IX contenga 4000_{16}. Dopo l'esecuzione di

SET 6,(IX + 5H)

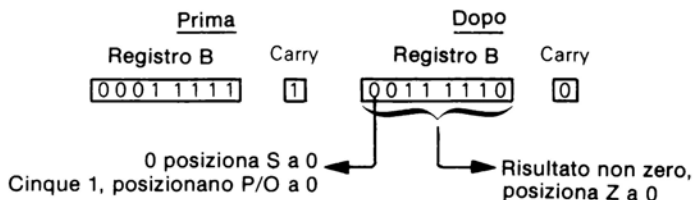
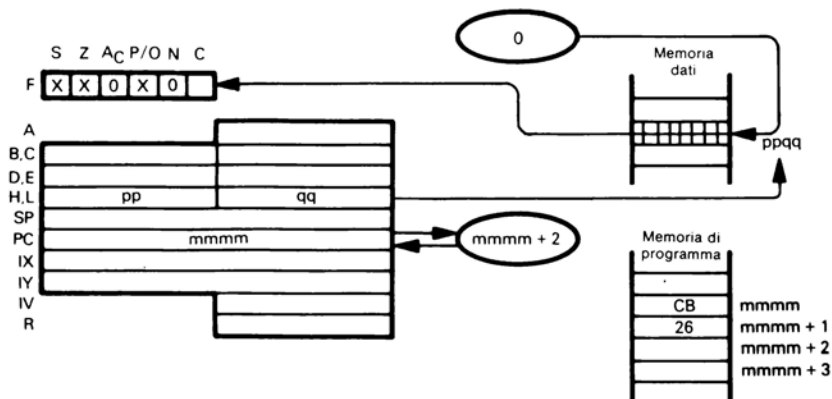
il bit 6 della locazione di memoria 4005_{16} sarà 1.

bbb è lo stesso di SET b,(HL)

Questa istruzione è identica a SET b,(IX + disp), tranne che essa usa il registro IY invece del Registro IX.

SLA reg — SPOSTA IL CONTENUTO DEL REGISTRO A SINISTRA IN MODO ARITMETICO

L'illustrazione mostra l'esecuzione di SLA C:

Sposta il contenuto del registro specificato a sinistra di un bit, posizionando a 0 il bit meno significativo.

Supponiamo che il Registro B contenga $1F_{16}$ e che il Carry=1. Dopo l'esecuzione di

SLA B

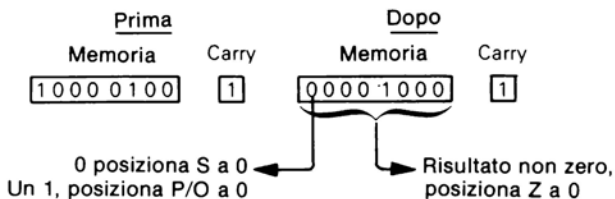
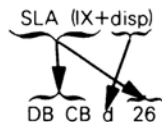
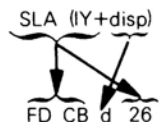
il Registro B conterrà $3E_{16}$ e il Carry sarà 0.

SLA (HL) — SPOSTA IL CONTENUTO DELLA LOCAZIONE SLA (IX +disp) DI MEMORIA A SINISTRA IN MODO ARITMETICO SLA (IY +disp)

L'illustrazione mostra l'esecuzione di SLA (HL):

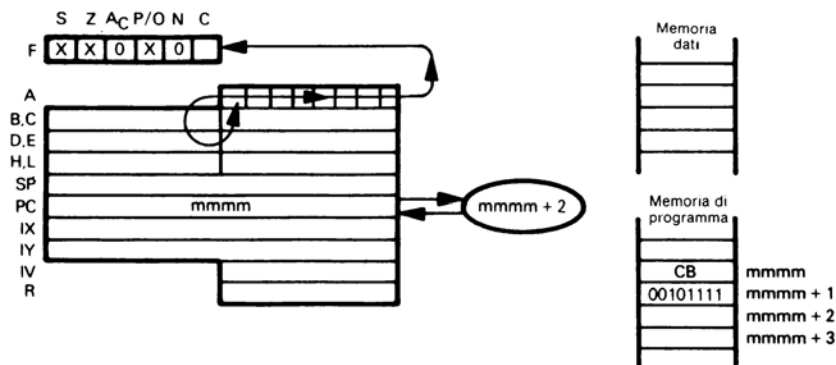
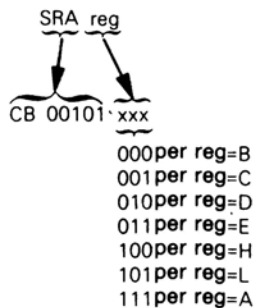
SLA (HL)
CB 26

Sposta il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) a sinistra di un bit, posizionando a 0 il bit meno significativo.

Supponiamo che la coppia di registri HL contenga 4500_{16}, che la locazione di memoria 4500_{16} contenga 84_{16} e che il Carry=0. Dopo l'esecuzione di

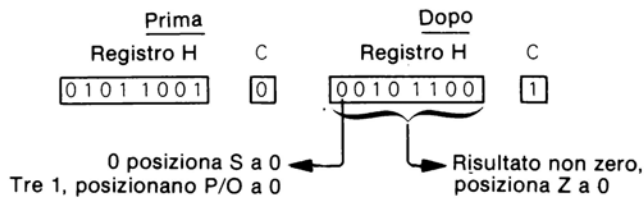
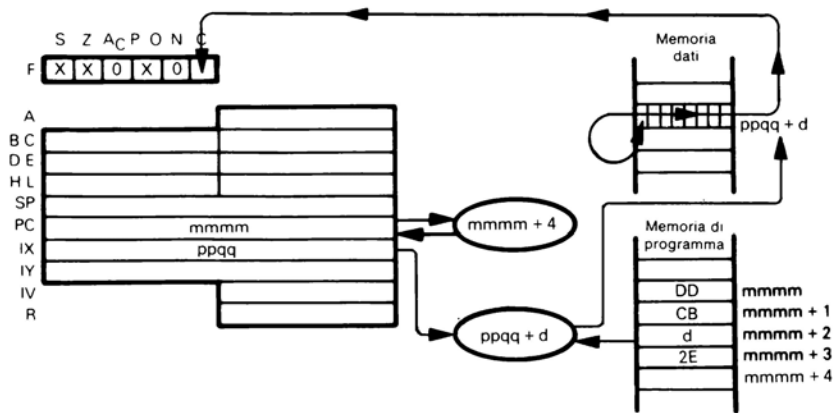
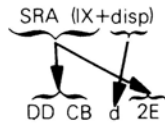
SLA (HL)

la locazione di memoria 4500_{16} conterrà 08_{16} e il Carry sarà 1.

Sposta il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del valore d del dislocamento) a sinistra di un bit in modo aritmetico, posizionando a 0 il bit meno significativo.

Questa istruzione è identica a SLA (IX + disp), ma usa il registro IY invece del registro IX.

SRA reg — SPOSTA IL CONTENUTO DEL REGISTRO A DESTRA IN MODO ARITMETICO

L'illustrazione mostra l'esecuzione di SRA A:

Sposta il registro specificato a destra di un bit. Il bit più significativo rimane immutato.

Supponiamo che il Registro H contenga 59_{16} e che Carry=0. Dopo l'esecuzione della istruzione

SRA H

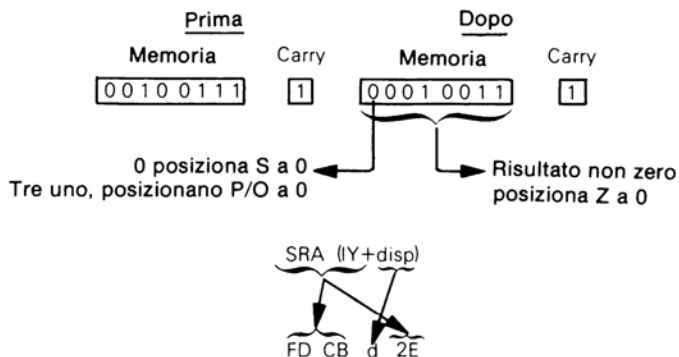
il registro H conterrà $2C_{16}$ e il Carry sarà 1.

SRA (HL) — SPOSTA A DESTRA IL CONTENUTO DELLA
SRA (IX +disp) POSIZIONE DI MEMORIA IN MODO ARITMETICO
SRA (IY +disp)

L'illustrazione mostra l'esecuzione di SRA (IX + disp):

Sposta il contenuto della locazione di memoria (specificata dalla somma del contenuto del Registro IX e del valore d del dislocamento) a destra. Il bit più significativo rimane immutato.

Supponiamo che il Registro IX contenga 3400_{16}, che la locazione di memoria $34AA_{16}$ contenga 27_{16} e che Carry=1. Dopo l'esecuzione di

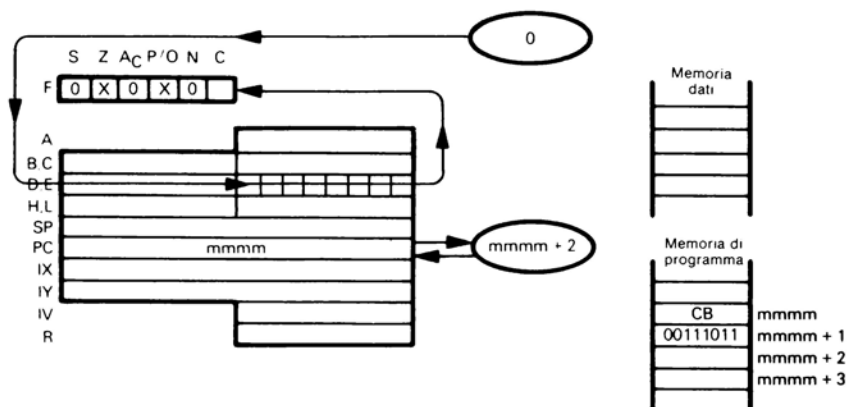
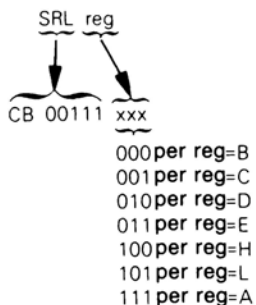
SRA (IX + 0AAH)

la locazione di memoria $34AA_{16}$ conterrà 13_{16} e il Carry sarà 1.

SRA (IY+disp)

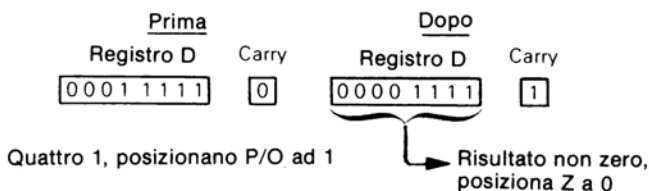
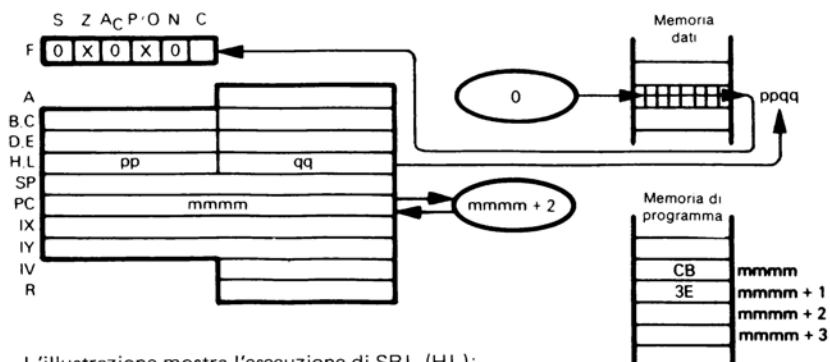
FD CB d 2E

Questa istruzione è identica a SRA (IX + disp), ma usa il registro IY invece del registro IX.

Sposta il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) a destra di un bit. Il bit più significativo rimane immutato.

SRL reg — SPOSTA IL CONTENUTO DEL REGISTRO A DESTRA IN MODO LOGICO

L'illustrazione mostra l'esecuzione di SRL E .

Sposta il contenuto del registro specificato a destra di un bit. Il bit più significativo è posizionato a 0.

Supponiamo che il Registro D contenga $1F_{16}$ e che Carry=0. Dopo l'esecuzione di

SRL D

il Registro D conterrà $0F_{16}$ e il Carry sarà 1.

SRL (HL) — SPOSTA IL CONTENUTO DELLA LOCAZIONE SRL (IX+disp) DI MEMORIA A DESTRA IN MODO LOGICO SRL (IY+disp)

L'illustrazione mostra l'esecuzione di SRL (HL):

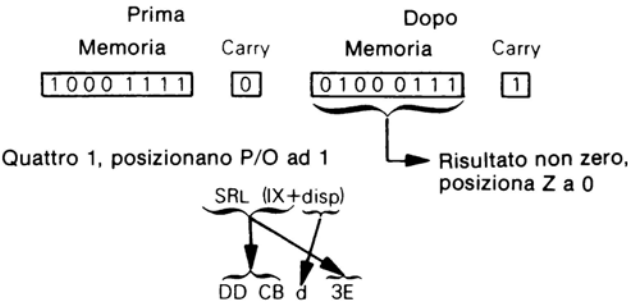
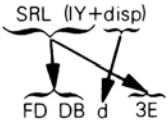
SRL (HL)

CB 3E

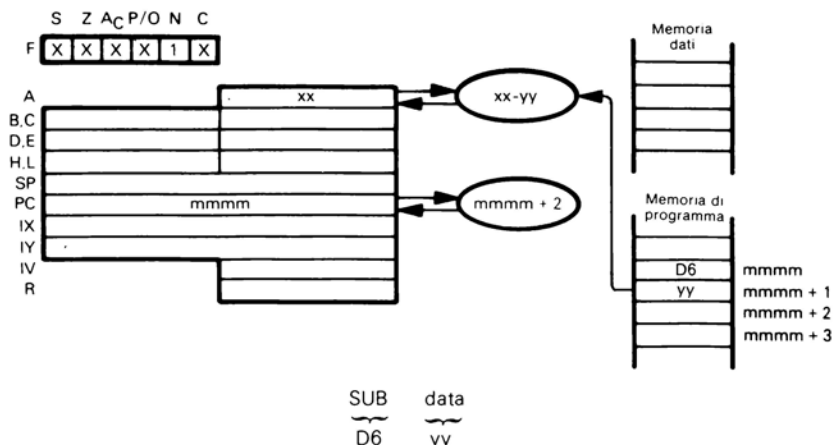
Sposta il contenuto della locazione di memoria (specificata dal contenuto della coppià di registri HL) a destra di un bit. Il bit più significativo è posizionato a 0.

Supponiamo che la coppia di registri HL contenga 2000_{16}, che la locazione di memoria 2000_{16} contenga $8F_{16}$ e che Carry=0. Dopo l'esecuzione di

SRL (HL)

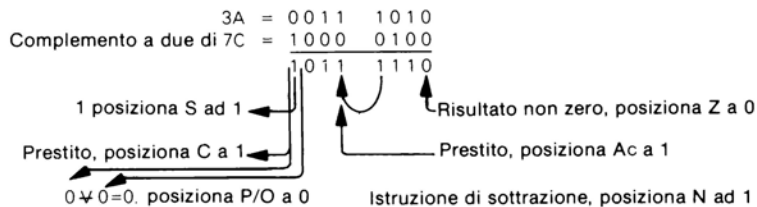
la locazione di memoria 2000_{16} conterrà 47_{16} e il Carry sarà 1.

Sposta il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del valore d del dislocamento) a destra di un bit. Il bit più significativo è posizionato a 0.

Questa istruzione è identica a SRL (IX + disp), ma usa il registro IY invece del registro IX.

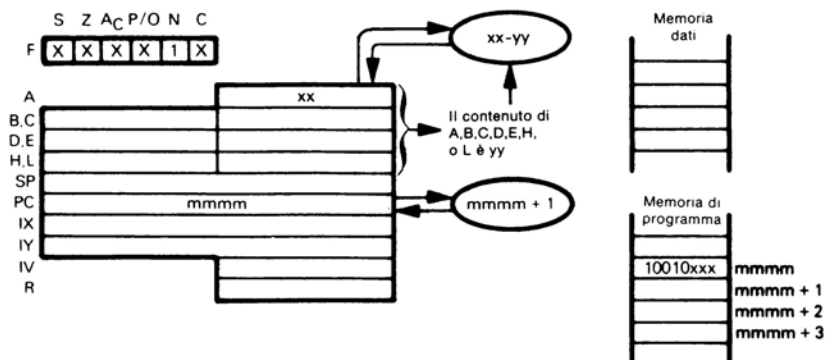
SUB data — SOTTRAZIONE IMMEDIATA DALL'ACCUMULATORE

SUB data
D6 yy

Sottrae il contenuto del secondo byte del codice oggetto dall'Accumulatore.

Supponiamo che xx=$3A_{16}$. Dopo l'esecuzione dell'istruzione

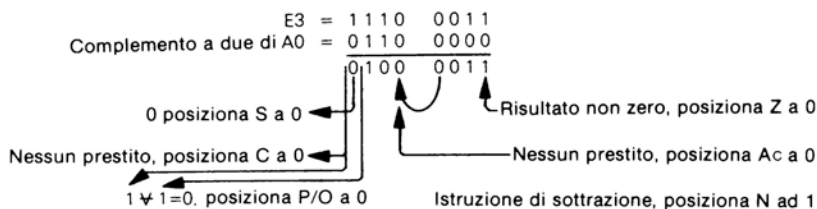
SUB 7CH

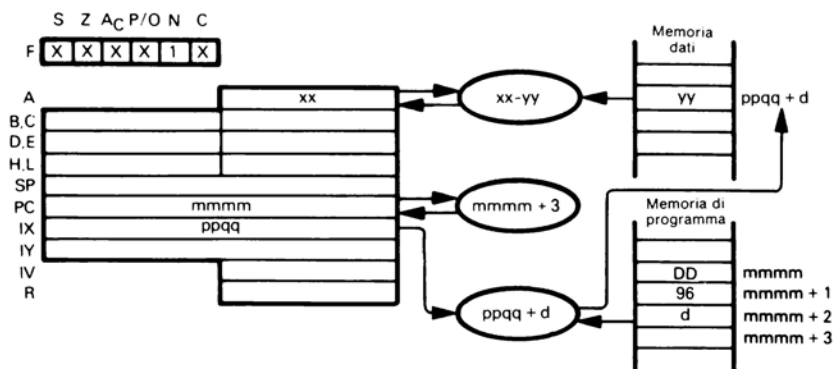
l'Accumulatore conterrà BE_{16}.

E' da notare che il carry risultante è complementato.

SUB reg — SOTTRAE IL REGISTRO DALL'ACCUMULATORE

Sottrae il contenuto del registro specificato dall'Accumulatore.

Supponiamo che xx=E3 e che il Registro H contenga $A0_{16}$. Dopo l'esecuzione di

SUB H

l'Accumulatore conterrà 43_{16}.

E' da notare che il carry risultante è complementato.

SUB (HL) — SOTTRAE LA MEMORIA DALL'ACCUMULATORE
SUB (IX + disp)
SUB (IY + disp)

L'illustrazione mostra l'esecuzione di SUB (IX + disp):

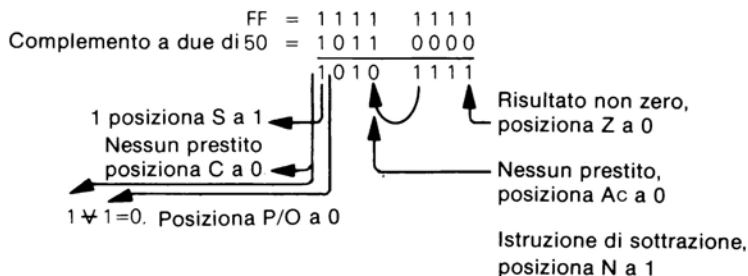
SUB (IX+disp)

DD 96 d

Sottrae il contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del valore d del dislocamento) dall'Accumulatore.

Supponiamo che ppqq=4000_{16}, xx=FF_{16} e che la locazione di memoria $40FF_{16}$ contenga 50_{16}. Dopo l'esecuzione di

SUB (IX + 0FFH)

l'Accumulatore conterrà AF_{16}.

E' da notare che il carry risultante è complementato.

SUB (IY+disp)

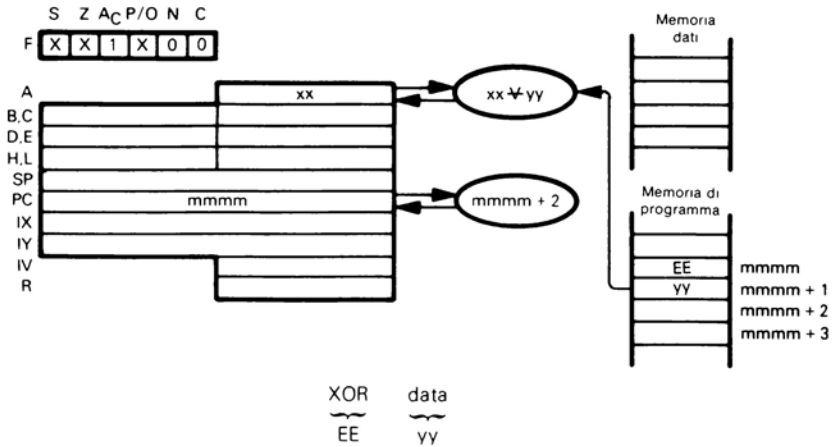
FD 96 d

Questa istruzione è identica a SUB (IX + disp), tranne che essa usa il registro IY invece del registro IX.

SUB (HL)

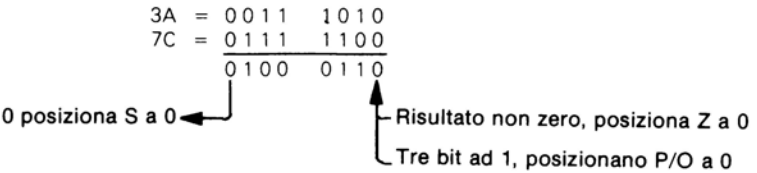
96

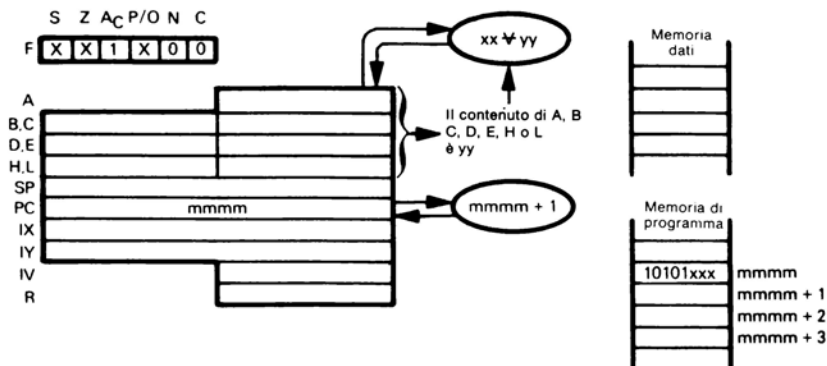
Sottrae il contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL), dall'Accumulatore.

XOR data – OR ESCLUSIVO IMMEDIATO CON L'ACCUMULATORE

Fa l'OR esclusivo del contenuto del secondo byte del codice oggetto con l'Accumulatore.

Supponiamo che xx=$3A_{16}$. Dopo l'esecuzione dell'istruzione

XOR 7CH

l'Accumulatore conterrà 46_{16}.

L'istruzione OR esclusivo è usata per provare cambiamenti negli stati dei bit.

XOR reg — OR ESCLUSIVO DEL REGISTRO CON L'ACCUMULATORE

```
XOR      reg
10101    xxx
         000 per reg=B
         001 per reg=C
         010 per reg=D
         011 per reg=E
         100 per reg=H
         101 per reg=L
         111 per reg=A
```

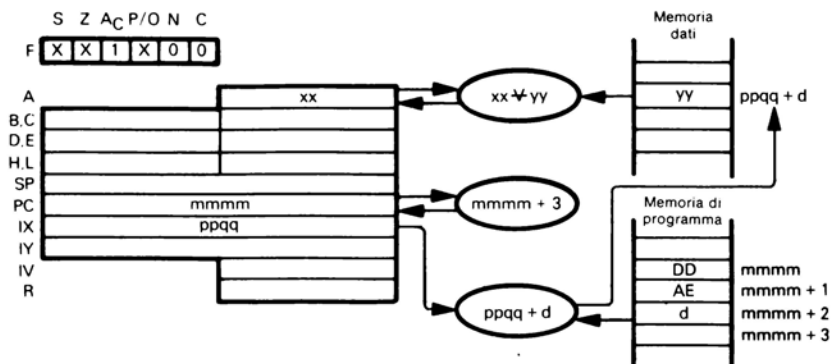
Fa l'OR esclusivo del contenuto del registro specificato con l'Accumulatore.

Supponiamo che xx=$E3_{16}$ e che il Registro E contenga $A0_{16}$. Dopo l'esecuzione della istruzione

XOR E

l'Accumulatore conterrà 43_{16}.

L'istruzione OR esclusivo è usata per verificare cambiamenti negli stati dei bit.

XOR (HL) — OR ESCLUSIVO DELLA MEMORIA CON
XOR (IX +disp) L'ACCUMULATORE
XOR (IY +disp)

L'illustrazione mostra l'esecuzione di XOR (IX + disp):

XOR (IX+disp)

DD AE d

Fa l'OR esclusivo del contenuto della locazione di memoria (specificata dalla somma del contenuto del registro IX e del valore d del dislocamento) con l'Accumulatore.

Supponiamo che xx=$E3_{16}$, ppqq=4500_{16} e che la locazione di memoria $45FF_{16}$ contenga $A0_{16}$. Dopo l'esecuzione dell'istruzione

XOR (IX + 0FFH)

l'Accumulatore conterrà 43_{16}.

```
E3 = 1110  0011
A0 = 1010  0000
     ----------
     0100  0011
```

0 posiziona S a 0

Risultato non zero, posiziona Z a 0

Tre bit ad 1, posizionano P/O a 0

XOR (IY+disp)

FD AE d

Questa istruzione è identica a XOR (IX + disp), tranne che essa usa il registro IY invece del registro IX.

XOR (HL)

AE

Fa l'OR esclusivo del contenuto della locazione di memoria (specificata dal contenuto della coppia di registri HL) con l'Accumulatore.

Capitolo 7

ALCUNI SOTTOPROGRAMMI USATI COMUNEMENTE

In molti programmi di un microcalcolatore capitano parecchie operazioni che non riguardano l'applicazione. Questo capitolo fornirà un certo numero di sequenze di istruzioni frequentemente usate.

Per fare l'uso più efficace di questo capitolo, dovreste studiare ogni sottoprogramma finchè non lo conoscerete così bene da modificarlo. Come semplice esercizio, dovreste tentare di riscrivere il sottoprogramma in modo che esso compia lo stesso lavoro usando un numero minore di cicli di esecuzione, o di istruzioni o di entrambi. Quindi riscrivete i programmi per implementare le variazioni. Per esempio, è illustrata la moltiplicazione binaria di numeri a 16 bit; che cosa si può dire su un sottoprogramma che moltiplica numeri a 32 bit? Guardate ogni esempio come una tipica sequenza di istruzioni illustrativa che voi probabilmente modificherete per soddisfare le vostre necessità.

I semplici programmi nel livello esposto in questo capitolo ricadono in queste quattro categorie:

1) Indirizzamento di memoria

2) Spostamento di dati

3) Aritmetica

4) Logica di sequenze di esecuzioni di programmi

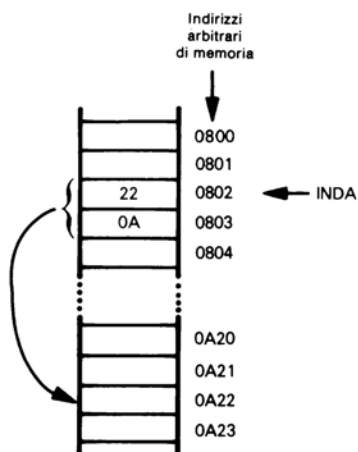
Descriveremo i programmi nella precedente sequenza di categorie.

INDIRIZZAMENTO DI MEMORIA

Lo Z80 ha una grande varietà non abituale di istruzioni di riferimento alla memoria: indirizzamenti diretti, indicizzati, impliciti e di incremento/decremento automatico sono tutti disponibili sullo Z80. Mostreremo come si possono implementare due altri modi di indirizzamento — indirizzamento indiretto e indirizzamento indiretto con successiva indicizzazione — mediante semplici sequenze di istruzioni. Entrambi questi modi sono descritti ed illustrati in An Introduction to Microcomputers: Volume I — Basic Concepts.

INDIRIZZAMENTO INDIRETTO

La CPU Z80 fornisce un indirizzamento indiretto ai registri dove una coppia di registri .(come HL) serve come puntatore ad una locazione di memoria. Tuttavia il vero indirizzamento indiretto di memoria specifica che l'indirizzo di memoria che si ri-

chiede sia immagazzinato in due byte di memoria:

Nella precedente illustrazione, i byte di memoria 0802_{16} e 0803_{16} contengono il richiesto indirizzo di memoria: $0A22_{16}$. In accordo col modo con cui lo stesso Z80 maneggia gli indirizzi a 16 bit, si mostra il byte di ordine minore dell'indirizzo prima del byte di ordine maggiore dell'indirizzo.

Tutto ciò che si richiede per simulare un indirizzamento indiretto come mostrato precedentemente è la seguente sequenza di istruzioni:

```
LD    HL,INDA      ; Carica l'indirizzo in HL
LD    A,(HL)       ; Carica il dato in A
```

La prima istruzione sposta l'indirizzo $0A22_{16}$ in HL. La seconda istruzione dimostra come accedere alla locazione di memoria $0A22_{16}$.

INDIRIZZAMENTO INDIRETTO, POST-INDICIZZATO

In alcune applicazioni è necessario o certamente preferibile effettuare un **indirizzamento indiretto post-indicizzato.** Usando l'indirizzamento indicizzato dello Z80, si **può realizzare un indirizzamento post-indicizzato nel seguente modo:**

```
LD    BC,(INDA)    ; Carica l'indirizzo indiretto in BC
ADD   IX,BC        ; Somma l'indirizzo indiretto all'indice
```

All'inizio di questa sequenza di istruzioni, supponiamo che l'indice sia il Registro Indice IX.

L'indice viene quindi sommato all'indirizzo indiretto e il risultato è messo nel registro Indice; si può ora effettuare qualunque operazione di memoria usando l'indirizzo Indice come indirizzo.

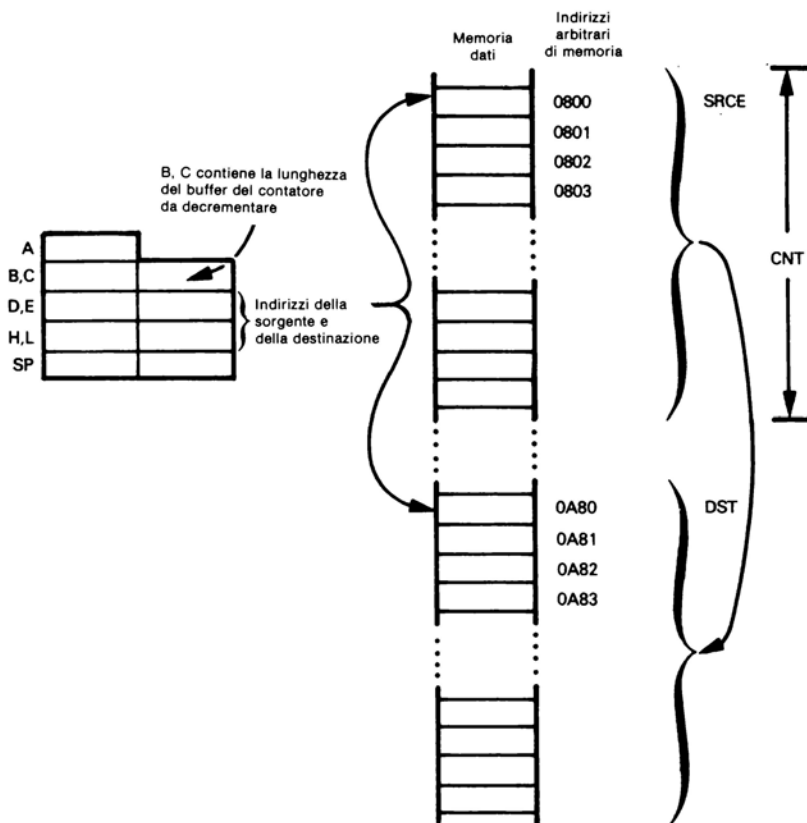
SPOSTAMENTO DI DATI

Esamineremo ora alcune sequenze di istruzioni che individuano e spostano blocchi contigui di byte di dati — buffer di dati di lunghezza qualsiasi.

SEMPLICE SPOSTAMENTO DI BLOCCHI DI DATI

Cominciando con un programma molto semplice, consideriamo lo spostamento del contenuto di un blocco contiguo di byte di memoria dati da un'area di memoria ad un'altra. Questa operazione è resa estremamente semplice dall'istruzione unica di trasferimento di un blocco fornita dalla CPU Z80. Le istruzioni di trasferimento di un blocco funzionano con tre coppie di registri:

HL indirizza la locazione sorgente
DE indirizza la locazione di destinazione
BC è un contatore di byte

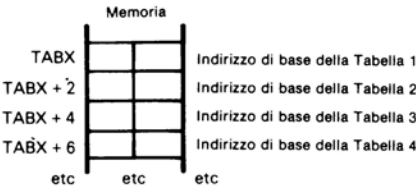
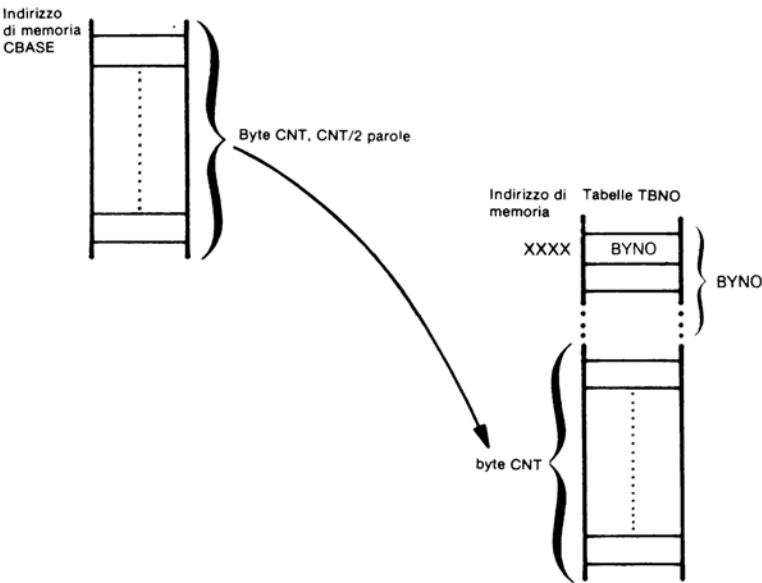
La seguente mappa di memoria illustra l'operazione di spostamento dei dati:

Questo è il programma di spostamento dei dati:

```
LD    HL,SRCE    ; Carica l'indirizzo della sorgente in HL
LD    DE,DST     ; Carica l'indirizzo della destinazione in DE
LD    BC,CNT     ; Carica il contatore dei byte in BC
LDIR             ; Trasferimento dei dati
```

La singola istruzione LDIR compie tutto il lavoro per noi — essa trasferisce il byte del dato puntato da HL alla locazione puntata da DE, quindi incrementa HL e DE al punto del successivo byte, decrementa il contatore BC e ripete il processo finchè il contatore non sia 0.

RICERCHE TABELLARI MULTIPLE

Consideriamo ora una ricerca tabellare multipla. Questa è una variazione più complessa dello spostamento di dati di quello appena descritto.

Gli indirizzi di partenza di un numero indefinito di tabelle di dati sono immagazzinati in una tabella indice. L'indirizzo di partenza della tabella indice è dato dalla label TABX:

Parecchi byte di dati sono memorizzati temporaneamente, a cominciare da una locazione di memoria identificata dalla label CBASE. Il numero reale di byte di dati si può trovare in una locazione di memoria identificata dalla label CNT. Questo buffer sorgente è equivalente al buffer sorgente nel programma di spostamento di dati appena descritto.

La destinazione del blocco di dati è una delle tabelle di dati. Il numero della tabella è identificato dal simbolo TBNO, che è caricato come dato immediato. I primi due byte di ogni tabella identificano il dislocamento per il primo byte libero della tabella; in altre parole supponiamo che ogni tabella sia riempita parzialmente e che il blocco di dati debba essere spostato nella fine non occupata della tabella selezionata. Il richiesto spostamento di dati può essere illustrato come nella figura in basso di pagina 7-4.

Ecco l'appropriata sequenza di istruzioni:

```
LD    HL,(TABX+TABNO)   ; Carica l'indirizzo di targa della tabella
                        ; in HL
LD    E,(HL)            ; Carica il dislocamento (BYNO) per il pri-
                        ; mo byte libero in DE
INC   HL
LD    D,(HL)
ADD   HL,DE             ; Somma ad HL, ottenendo l'indirizzo del
                        ; primo byte libero
EX    DE,HL             ; Sposta l'indirizzo in DE
LD    HL,CBASE          ; Carica l'indirizzo di base d'ingresso nel
                        ; buffer (CBASE) in HL
LD    BC,(CNT)          ; Carica il contatore dei byte in BC
LDIR                    ; Trasferisci i dati
```

CLASSIFICAZIONE DEI DATI

Entrambi gli esempi di programmazione che abbiamo descritto così semplicemente spostano un blocco di dati da una locazione ad un'altra. E' pure importante la riorganizzazione dei dati; perciò **illustreremo un programma di classificazione.**

La classificazione, come illustrato, prende una sequenza di numeri binari con segno immagazzinati in locazione di memoria contigue, e li riorganizza in ordine ascendente cosicchè il numero minore diventi il primo e il numero maggiore diventi l'ultimo.

Il programma di classificazione che programmeremo usa un algoritmo di tipo "bubble-up". Consideriamo una sequenza di numeri tale che l'etichetta LIST identifica l'indirizzo della locazione di memoria del primo numero. I passi del programma di classificazione necessari sono:

CLASSIFICAZIONE DI DATI

1) Fai un passo all'inizio di LIST ed inizializza un flag per indicare una condizione "nessun scambio".
2) Confronta una coppia consecutiva di numeri. Se il primo numero è minore del secondo numero non fare niente; altrimenti scambia i due numeri e posiziona il flag per indicare "scambio effettuato".
3) Confronta l'indirizzo del secondo numero con la fine dell'indirizzo della lista, identificata dall'etichetta ENDL. Se non è nella fine, incrementalo cosicchè il secondo numero della coppia corrente diventi il primo numero della coppia successiva e ritorna al passo 2.
4) Alla fine della lista controlla il flag di "scambio". Se durante il passo non è stato effettuato nessuno scambio, tornare al passo 1 per compiere un altro passo.
5) Se si è compiuto un passo senza alcuno scambio, tutti i numeri sono in ordine. Uscita.

Come esempio, consideriamo il caso che i numeri da 1 a 10 siano in ordine inverso. Durante il primo passo saranno effettuati nove scambi, e alla fine di esso i numeri

maggiori saranno stati messi "in alto" (bubbled up):

	START	DOPO IL PASSO 1
LIST	10	9
	9	8.
	8	7
	7	6
	6	5
	5	4
	4	3
	3	2
	2	1
ENDL	1	10

Saranno necessari altri otto passi per mettere in ordine tutti i numeri e sarà poi necessario un decimo passo per ottenere una condizione di uscita di "nessuno scambio".

SORT è implementato come un sottoprogramma; prima della chiamata del sottoprogramma, si carica HL con l'indirizzo di inizio (LIST) del dato che deve essere classificato e B con la lunghezza della lista.

```
        LD   HL,LIST
        CALL SORT
        —
        —
        —
SORT:   LD   (SVAD),HL      ; Salva l'indirizzo di LIST
LOOP1:  LD   HL,(SVAD)
        LD   B,ENDL-LIST
        RES  0,D            ; Inizializza l'indicatore di "scambio"
LOOP2:  LD   A,(HL)         ; Carica il primo byte nell'Accumulatore
        INC  HL             ; Punta al prossimo byte
        CP   A,(HL)         ; Confronta i due byte
        JR   NC,SORT1
        LD   E,(HL)         ; Le prossime cinque istruzioni fanno lo scambio
        LD   (HL),A
        DEC  HL
        LD   (HL),E
        INC  HL
        SET  0,D            ; Posiziona il flag di "scambio"
SORT1:  DJNZ                ; Ripete il loop se la lista non è esaminata a
                            ; fondo
        BIT  0,D            ; Controllo sugli scambi
        JR   NZ,LOOP1       ; Ritorno se non ci sono scambi
        RET
```

ARITMETICA

In questo gruppo si descriveranno l'addizione, la sottrazione, la moltiplicazione e la divisione. Le funzioni trascendenti sono abbastanza complesse da richiedere a dei libri di testo dedicati a questo soggetto, così non cominceremo neppure l'argomento.

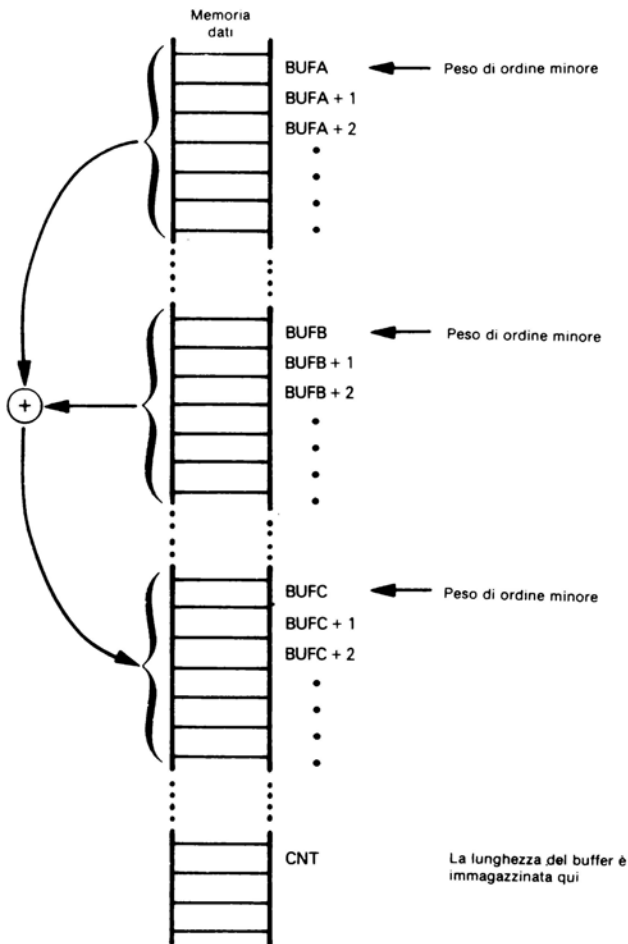
Anche entro i semplici confini dell'addizione, della sottrazione, della moltiplicazione e della divisione c'è un certo grado di larghezza di veduta che va oltre lo scopo della materia che si può esporre. Si richiedono algoritmi significativamente differenti,

dipendentemente dalla grandezza del numero. L'aritmetica binaria e decimale richiede algoritmi differenti. Perciò, **per l'addizione e la sottrazione considereremo numeri binari o decimali grandi o piccoli. Per la moltiplicazione e la divisione considereremo solamente numeri binari piccoli.**

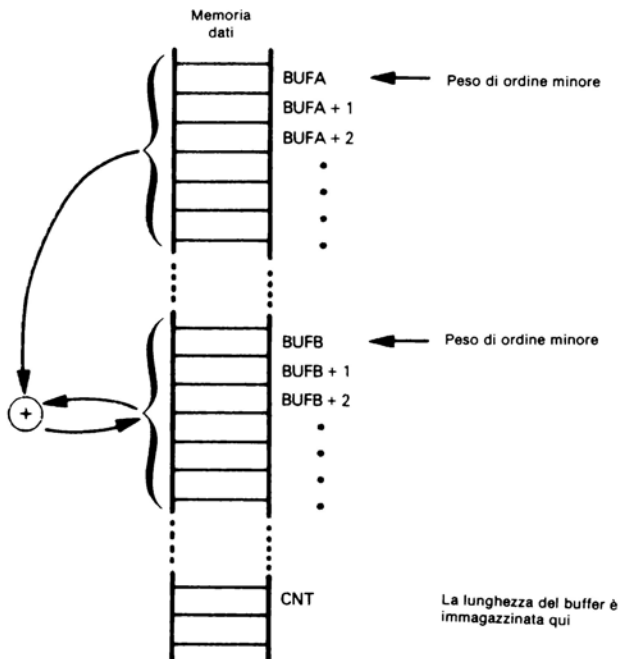
ADDIZIONE BINARIA

Consideriamo dapprima l'addizione binaria multibyte.

Si debbano addizionare due numeri interi positivi ciascuno dei quali sia lungo CNT byte. Gli indirizzi di partenza dei buffer dei numeri sia dato da BUF1 e da BUF2. La risposta deve essere immagazzinata in un buffer che inizi in BUF3.

L'addizione multibyte può essere illustrata come segue:

Questa sequenza di istruzioni effettua la addizione illustrata:

```
         LD    A,(CNT)     ; Carica la lunghezza del buffer e lo salva
         LD    B,A         , in B
         LD    HL,BUFC     ; Carica l'indirizzo del buffer della risposta in HL
         PUSH  HL          ; Salvataggio sullo stack
         LD    DE,BUFA     ; Carica l'indirizzo del primo buffer in DE
         LD    HL,BUFB     ; Carica l'indirizzo del secondo buffer in HL
         AND   A           ; Azzera il Carry
LOOP     LD    A,(DE)      ; Carica il successivo byte di BUFA
         ADC   (HL)        ; Somma il successivo byte di BUFB
         EX    (SP),HL     ; Salvataggio nel successivo byte del buffer della
         LD    (HL),A      ; risposta
         INC   HL          ; Incrementa l'indirizzo di BUFC
         EX    (SP),HL
         INC   DE          ; Incrementa l'indirizzo di BUFA
         INC   HL          ; Incrementa l'indirizzo di BUFB
         DJNZ  LOOP        ; Decrementa il contatore e ritorna per altri byte
                           ; se non è zero
```

L'addizione multibyte è più semplice se si può immagazzinare la somma in uno dei buffer della sorgente:

Ecco la sequenza di istruzioni più breve:

```
         LD    A,(CNT)     ; Carica la lunghezza del buffer e lo salva
         LD    B,A
         LD    DE,BUFA     ; Carica l'indirizzo del primo byte in DE
```

```
        LD    HL,BUFB    ; Carica l'indirizzo del secondo byte in HL
        AND   A          ; Azzera il Carry
LOOP    LD    A,(DE)     ; Carica il successivo byte di BUFA
        ADC   (HL)       ; Somma il successivo byte in BUFB
        LD    (HL)       ; Somma il successivo byte di BUFB
        LD    (HL),A     ; Immagazzina la risposta
        INC   DE         ; Incrementa l'indirizzo di BUFA
        INC   HL         ; Incrementa l'indirizzo di BUFB
        DJNZ             ; Decrementa il contatore e ritorna se non è zero
```

SOTTRAZIONE BINARIA

La sottrazione binaria è quasi identica all'addizione binaria, perchè lo Z80 ha istruzioni di sottrazione speciali. Nell'uno e nell'altro sottoprogramma, sostituite semplicemente l'istruzione ADC con l'istruzione SBC ed il risultato sarà una sottrazione binaria corretta.

ADDIZIONE DECIMALE

Anche l'addizione decimale è molto facile usando un microcalcolatore Z80. **Inserite semplicemente un'istruzione DAA dopo l'istruzione ADC** nell'uno e nell'altro programma dell'addizione binaria ed otterrete l'addizione decimale.

```
        –
        –
        –
LOOP    LD    A,(DE)     ; Carica il successivo byte di BUFA
        ADC   (HL)       ; Somma il successivo byte di BUFB
        DAA              ; Aggiustamento decimale del risultato
        LD    (HL),A     ; Immagazzina la risposta
        –
        –
        –
```

Occorre, tuttavia, una precauzione: il programma di addizione decimale che creerete presuppone di memorizzare nei buffer della sorgente dati decimali validi codificati in binario. Se per errore voi avete dati non validi in uno o nell'altro dei buffer della sorgente, genererete una risposta senza significato – e non la conoscerete.

Se il vostro programma non può garantire che i dati nei buffer della sorgente siano dati decimali validi in codifica binaria, allora dovete scrivere un programma per controllare il contenuto del buffer e assicurarvi che nessuna unità di 4 bit alta o bassa in un byte qualsiasi contenga un codice binario da A a F.

SOTTRAZIONE DECIMALE

Poichè lo Z80 ha uno speciale flag di Subtract (N), **l'istruzione Decimal Adjust Accumulator (DAA) può essere pure usata per la sottrazione decimale. Inserite semplicemente una istruzione DAA dopo l'istruzione SBC ed otterrete la sottrazione decimale.** Qui si applica la stessa precauzione menzionata per l'addizione decimale: dovete assicurarvi che nei buffer della sorgente siano immagazzinati dati decimali validi in codifica binaria.

MOLTIPLICAZIONE E DIVISIONE

Nei sistemi a microcalcolatore ci si deve avvicinare alla moltiplicazione ed alla divisione con precauzione. Queste sono operazioni che non si adattano alla organizzazione

di un microcalcolatore; una qualunque moltipliczione o divisione di una certa importanza può occupare un tempo così lungo che degraderà severamente la prestazione totale. **Se la vostra applicazione col microcalcolatore farà largo uso di moltiplicazioni, divisioni o di funzioni trascendenti, dovreste considerare seriamente l'uso di uno dei molti chip "calcolatore/aritmetica" che sono attualmente disponibili commercialmente.** Il trasferimento di aritmetica complessa su tali chip può rendere la differenza per un sistema a microcalcolatore vitale o non vitale nella vostra applicazione.

Potete implementare semplici moltiplicazioni e divisioni in sistemi a microcalcolatore che non fanno uso esteso o che consuma tempo di questi programmi; descriveremo, perciò alcune semplici sequenze di programmi.

MOLTIPLICAZIONE BINARIA A 8 BIT

Consideriamo la moltiplicazione di due valori a 8 bit senza segno per generare un prodotto a 16 bit. Il modo più semplice per ottenere questa moltiplicazione è di addizionare a 0 il moltiplicatore il numero di volte dato dal moltiplicando. Per esempio, **potete moltiplicare 4 per 3 addizionando tre volte 4 a 0.**

Supponiamo che il Registro B contenga il moltiplicando e che il Registro E contenga il moltiplicatore. Il seguente programma effettua l'operazione di moltiplicazione, riportando il risultato a 16 bit nell'Accumulatore A (ordine minore) e nel Registro C (ordine maggiore)

```
        LD   A,0        ; Azzera A e C per
        LD   C,A        ; inizializzare il buffer della risposta
        CP   B          ; Test se c'è 0 in B (moltiplicando)
        RET  Z          ; Se è 0, la risposta è 0, così finisce
LOOP    ADD  E          ; Addiziona il moltiplicatore al byte di ordine mino-
                        ; re della risposta
        JR   NC,NEXT    ; Se il Carry è posizionato
        INC  C          ; Incrementa C (byte di ordine maggiore)
NEXT    DJNZ LOOP       ; Decrementa il moltiplicando. Se non è zero salta
                        ; nuovamente ad ADD
        RET             ; Ritorno quando la moltiplicazione è completata
```

Questo programma potrebbe essere molto veloce (se il moltiplicando è 0, per cui si eseguono solo quattro istruzioni) o molto lenta (se il moltiplicando è 255, per cui questo programma potrebbe portar via 1025 esecuzioni di istruzioni).

In generale c'è un modo più veloce per eseguire delle moltiplicazioni. Usando la notazione decimale, consideriamo la seguente moltiplicazione:

```
      142    Moltiplicando
  x   317    Moltiplicatore
  -------
      994  \ prodotti
     142    } parziali
    426    /
  -------
    45014    Prodotto
```

Questo è il modo che abbiamo imparato per fare moltiplicazioni usando carta e matita. Ogni prodotto parziale è uguale al moltiplicando moltiplicato da un peso del moltiplicatore. Abbiamo cominciato a moltiplicare il moltiplicando (142) per il peso (7) più a destra del moltiplicatore. Successivamente abbiamo moltiplicato (142) per il secondo peso (1) del moltiplicatore. Il risultato parziale di questa moltiplicazione è spostato a sinistra di una posizione. Si è quindi usato il peso del moltiplicatore più a sinistra per moltiplicare 142 e il risultato parziale è stato spostato a sinistra di una

posizione in più. Dopo l'effettuazione di tutte le operazioni di moltiplicazione, si sono sommati insieme i prodotti parziali per ottenere il prodotto finale. **Questo metodo si segue bene con operazioni di carta e matita; tuttavia, non è il metodo più efficiente per ottenere una moltiplicazione con un calcolatore. Guardiamo un altro metodo.**

Anzitutto, non si deve aspettare che si siano completate tutte le moltiplicazioni prima di sommare insieme i prodotti parziali; si può generare "un totale veloce" o un risultato intermedio addizionando immediatamente ogni prodotto parziale al precedente prodotto parziale. Per esempio:

142	Moltiplicando
317	Moltiplicatore
000	risultato intermedio (condizione iniziale)
+ 994	prodotto parziale (7 x 142)
994	risultato intermedio
+ 142	prodotto parziale (1 x 142)
2414	risultato intermedio
+ 426	prodotto parziale (3 x 142)
45014	Prodotto

Sebbene questo metodo impieghi più tempo quando si usi carta e matita, esso è il metodo di moltiplicazione più efficiente per un calcolatore.

Ora, **pure noi possiamo fare in modo che ogni prodotto parziale sia spostato a sinistra di un peso prima di addizionare il risultato intermedio.** Ci sono due modi per fare ciò: possiamo effettivamente spostare a sinistra il prodotto parziale o possiamo spostare il prodotto intermedio a destra — l'effetto sarà lo stesso. Ritardiamo per il momento la decisione su questo punto, per considerare un'altra possibilità.

Sebbene abbiamo imparato ad effettuare una moltiplicazione cominciando dal bit meno significativo (il più a destra) del moltiplicatore, non c'è nulla che ci impedisca dal cominciare dall'altra parte dal momento che si segue il significato del peso moltiplicante usato. Per esempio:

142	Moltiplicando
317	Moltiplicatore
000	risultato intermedio (condizione iniziale)
+426	prodotto parziale (3 x 142)
426	risultato intermedio
+ 142	prodotto parziale (1 x 142)
4402	risultato intermedio
+ 994	prodotto parziale (7 x 142)
45014	Prodotto

Notiamo in questo esempio che, quando cominciamo col peso più significativo del moltiplicatore, i prodotti parziali successivi devono essere spostati a destra (invece che a sinistra) prima di essere addizionati. Di nuovo, lo spostamento del prodotto parziale potrebbe essere pure effettuato spostando il risultato intermedio nella direzione opposta.

Riassumendo, possiamo iniziare una operazione di moltiplicazione sia col peso più significativo che col peso meno significativo del moltiplicatore, e possiamo spostare sia i prodotti parziali che i risultati intermedi per ottenere l'allineamento appropriato dei pesi significativi.

Quale metodo useremo? Prima di decidere, vediamo che cosa accade per la moltiplicazione di numeri binari. **Poichè un peso binario è limitato ad avere valori 0 od 1, ciò significa che la moltiplicazione a livello di un solo peso degenera in una addizione o in nessun addizione.** Cioè:

Moltiplicando:	1011	1011	
Peso del moltiplicatore:	x 1	x 0	
Risultato intermedio:	0000	0000	
Prodotto parziale:	+1011	+0000	(non necessita alcuna
Nuovo risultato intermedio:	1011	0000	addizione)

Con ciò in mente, diamo un'altra occhiata ai metodi di moltiplicazione che abbiamo discusso. Dapprima, **si può vedere che non occorrono più passi separati per l'operazione di moltiplicazione e la successiva addizione del prodotto parziale del risultato intermedio;** il moltiplicare il moltiplicando per 1 è semplice come addizionare il moltiplicando al risultato intermedio.

Ora, **poichè stiamo effettuando operazioni di addizione invece di operazioni di moltiplicazione e di addizione, non dobbiamo maneggiare prodotti parziali** — possiamo addizionare semplicemente il moltiplicando direttamente al risultato intermedio. Se eliminiamo il prodotto parziale, allora vogliamo effettuare l'operazione di spostamento sul risultato intermedio. **Scriviamo ora due insiemi di regole di moltiplicazione di numeri binari.**

Metodo N. 1:

a) Sposta immediatamente il risultato un posto a destra.

b) Se il peso meno significativo è zero, salta il passo c è vai al passo d.

c) Somma il moltiplicando al risultato intermedio.

d) Ripeti i passi a, b e c per il digit successivo (più significativo) del moltiplicatore finchè non si siano usati tutti i digit.

Metodo N. 2:

a) Sposta immediatamente il risultato un posto a sinistra.

b) Se il peso più significativo è zero, salta il passo c e vai al passo d.

c) Somma il moltiplicando al risultato intermedio.

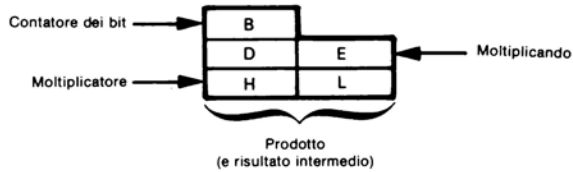
d) Ripeti i passi a, b e c per il digit successivo (meno significativo) del moltiplicatore finchè non si siano usati tutti i digit.

Ora che abbiamo esaminato il meccanismo usato nella moltiplicazione di numeri binari e sviluppato alcuni insiemi di regole, vediamo come si possa implementare questi algoritmi usando lo Z80.

UN PROGRAMMA DI MOLTIPLICAZIONE BINARIA AD 8 BIT

Scriveremo ora un programma che moltiplicherà due valori ad 8 bit senza segno per generare un prodotto a 16 bit.

Consideriamo dapprima l'assegnazione dei registri: abbiamo bisogno di un registro ad 8 bit per il moltiplicatore, un registro ad 8 bit per il moltiplicando, un registro a 16 bit per il prodotto ed un registro da usare come contatore dei bit durante l'operazione di moltiplicazione.

Assegneremo i registri come segue:

Ora, alcune asegnazioni di registri possono sembrare un po' strane, specialmente ponendo il moltiplicatore nel registro H ed assegnando pure il registro H per il byte più significativo del prodotto. Tuttavia, procederemo a scrivere il nostro programma e le ragioni delle assegnazioni dei registri fatte sopra avranno quindi maggior significato.

Ecco il programma:

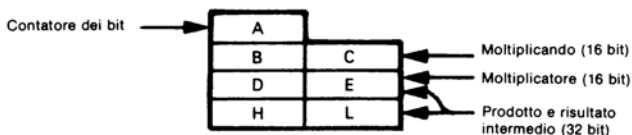
```
MULT:   LD   B,8        ; Carica il contatore in B
        LD   D,0        ; Azzera il registro D
        LD   L,D        ; Azzera il registro L
LOOP:   ADD  HL,HL      ; Sposta HL di un posto a sinistra
        JR   NC,DECB    ; Se non c'è nessun riporto, il moltiplicatore dei bit
                        ; è 0, saltare ADD
        ADD  HL,DE      ; Addiziona il moltiplicando al risultato intermedio
DECB:   DJNZ LOOP       ; Decrementa il contatore B. Se non è zero, ripetere
                        ; il loop
        RET             ; Ritorno
```

Abbiamo usato il "Metodo N. 2" della nostra discussione precedente per questo programma. Il programma è scritto come sottoprogramma e suppone che, prima di entrare in esso, il registro E contenga il moltiplicando ad 8 bit e che il registro H contenga il moltiplicatore ad 8 bit. **Confrontando il programma col "Metodo N. 2", esso sembrerà molto semplice, con la possibile eccezione della prima istruzione ADD.** Perchè si addiziona HL a sé stesso? La risposta non ovvia è che si usa realmente la istruzione ADD per spostare i registri H ed L di un bit a sinistra. (Addizionando un numero binario a sé stesso si ha come risultato uno spostamento di una posizione a sinistra di un bit del numero stesso). Ora, sembrerà più immediato usare semplicemente una istruzione di spostamento invece che una istruzione ADD. Tuttavia, l'insieme di istruzioni dello Z80 non fornisce istruzioni per effettuare operazioni di spostamento su una coppia di registri a 16 bit. Perciò, dovremo usare due istruzioni di spostamento per realizzare la stessa cosa della sola istruzione ADD.

Notiamo che quando si sposta la coppia di registri HL a sinistra, si realizzano due cose. Effettuiamo lo spostamento a sinistra del risultato intermedio come richiesto dal nostro algoritmo di moltiplicazione e spostiamo inoltre il bit più significativo del moltiplicatore nel flag di Riporto. L'istruzione JUMP che segue ADD verifica quindi se il bit del moltiplicatore è stato spostato ad 1 o a 0.

Come spostiamo HL a sinistra così spostiamo il moltiplicatore, in modo che la coppia di registri possa essere usata per il risultato intermedio. Dopo aver girato otto volte nel loop, il moltiplicatore sarà stato spostato completamente fuori del registro H e HL ora conterrà il prodotto a 16 bit.

MOLTIPLICAZIONE BINARIA A 16 BIT

Consideriamo ora la moltiplicazione di due numeri a 16 bit, producendo un risultato a 32 bit. L'algoritmo che useremo è lo stesso usato per la moltiplicazione a 8 bit; tuttavia, si richiederanno alcune istruzioni addizionali per manipolare i registri. **Ecco le assegnazioni dei registri:**

La coppia di registri DE conterrà il moltiplicatore a 16 bit all'inizio del programma e conterrà i 16 bit più significativi del prodotto a 32 bit quando sarà completata la moltiplicazione.

```
MPY:    LD   HL,0000H  ; Inizializzazione del prodotto parziale con HL a zero
        LD   A,16      ; Inizializzazione del contatore
LOOP:   ADD  HL,HL     ; Spostamento del risultato parziale a sinistra nel Carry
        EX   DE,HL     ; Scambio di DE con HL
        ADC  HL,HL     ; Spostamento del moltiplicatore a sinistra nel Carry
        EX   DE,HL     ; Ritorno del moltiplicatore spostato in DE
        JR   NC,DECA   ; Salto se non c'è nessuna addizione (bit del moltipli-
                       ; catore nel Carry uguale a 0)
        ADD  HL,BC     ; Addiziona il moltiplicando in BC al prodotto par-
                       ; ziale in HL
        JR   NC,DECA   ; Salto se non c'è nessun Carry dall'addizione
        INC  DE        ; Incremento di DE per propagare il Carry da ADD
DECA:   DEC  A         ; Decremento del contatore
        JP   NZ,LOOP   ; Se non è zero tornare a loop
        RET            ; Ritorno
```

DIVISIONE BINARIA

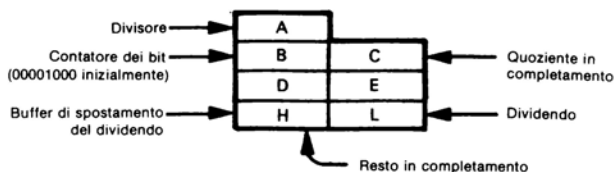
Il procedimento usato per ottenere una divisione binaria è molto simile a quello usato per la moltiplicazione. Qui il processo coinvolge la sottrazione piuttosto che l'addizione.

Consideriamo una semplice divisione a 8 bit. $B3_{16}$ diviso per 15_{16} può essere illustrato come segue:

```
                     1000      Quoziente
Divisore    10101 ) 10110011   Dividendo
                    10101
                     1011      Resto
```

Il risultato è 8_{16} con resto B_{16}.

L'algoritmo della divisione funziona spostando il dividendo in un registro che inizialmente è azzerato. Ogni volta che il contenuto del buffer di spostamento del dividendo è uguale o maggiore del divisore, il divisore viene sottratto dal contenuto del buffer di spostamento e si inserisce un peso binario 1 nella posizione appropriata dei bit del quoziente.

Consideriamo le seguenti assegnazioni di registri:

Supponiamo, inizialmente, che il divisore sia nel registro A e che il dividendo sia nel registro L. Il quoziente sarà generato nel registro C. **Ecco il programma di divisione risultante:**

```
DIV:    LD   BC,0800H   ; Caricamento del contatore dei bit ed azzeramento
                        ; del registro del quoziente
        LD   H,C        ; Azzeramento del buffer di spostamento del divi-
                        ; dendo (H)
        LD   E,H        ; Caricamento di zero nel registro E
        LD   D,A        ; Copia del divisore nel registro D
LOOP:   ADD  HL,HL      ; Spostamento del dividendo a sinistra nel registro H
        LD   A,H        ; Copia del buffer di spostamento del dividendo nel
                        ; registro A
        CP   D          ; Confronta il buffer di spostamento del dividendo
                        ; col divisore
        JR   C,NEXT     ; Se il dividendo è minore del divisore, non fare la
                        ; sottrazione
        SBC  HL,DE      ; Sottrazione del divisore dal buffer di spostamento
                        ; del dividendo
NEXT    CCF             ; Complemento del flag di Carry
        RL   C          ; Spostamento di un 1 o di uno 0 (dal Carry) nel
                        ; quoziente
        DJNZ LOOP       ; Decremento del contatore e ripetizione del loop fi-
                        ; no al suo completamento
        RET             ; Ritorno al programma chiamante
```

Alla fine, il quoziente è nel registro C e il resto è nel registro H.

Notiamo che abbiamo usato ancora una volta l'istruzione ADD per ottenere uno spostamento a sinistra della coppia di registri a 16 bit HL. Abbiamo inoltre usato l'istruzione di sottrazione a 16 bit (SBC); tuttavia, poichè abbiamo inizialmente posizionato a zero il contenuto del registro E, in realtà usiamo l'istruzione SBC semplicemente per sottrarre il contenuto del registro D dal contenuto del registro H — operazione di sottrazione a 8 bit. Abbiamo usato la versione a 16 bit dell'istruzione di sottrazione per ridurre il numero di istruzioni di spostamento tra registri altrimenti richiesto, poichè le istruzioni di sottrazione ad 8 bit richiedono l'uso del registro A, già usato.

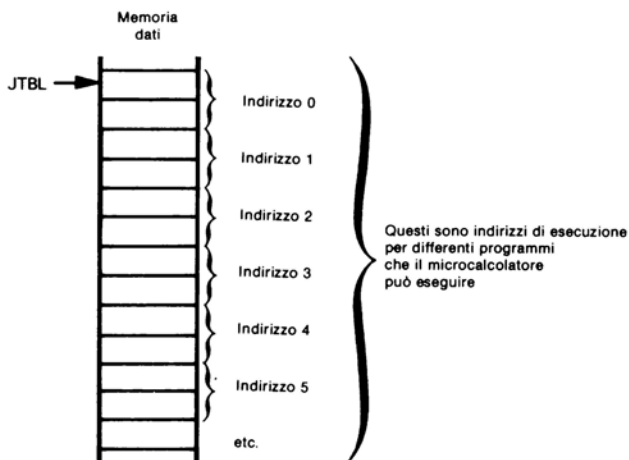
LOGICA DELLA SEQUENZA DI ESECUZIONE DI UN PROGRAMMA

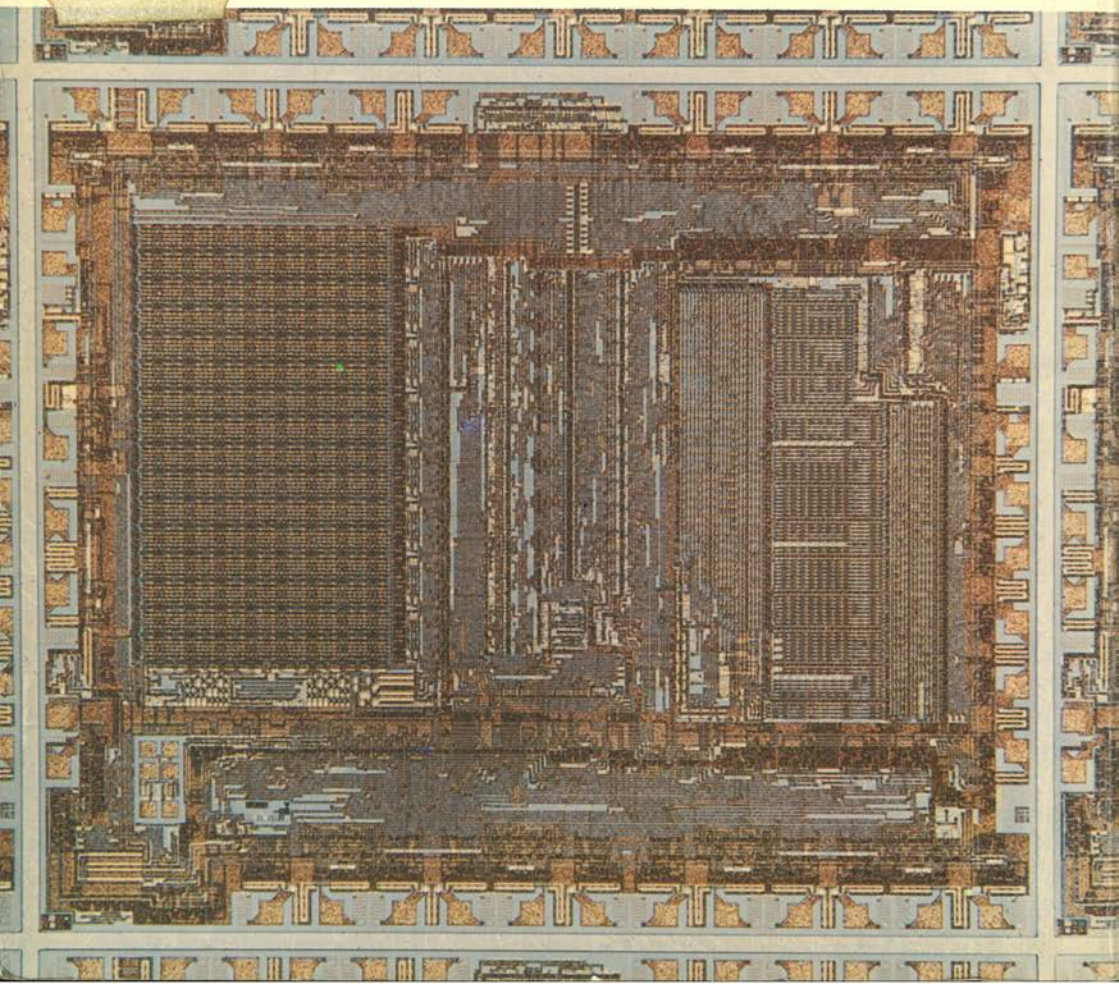
LA TABELLA DEI SALTI

In realtà c'è una sola sequenza di programma che necessita di essere descritta con questo titolo; essa è la tabella dei salti.

Ricordiamo che l'insieme delle istruzioni dello Z80 è ricco di istruzioni condizionate; le istruzioni Jump, Call e Return hanno tutte otto varianti condizionate, il che significa che non si richiedono speciali programmi quando la vostra logica può andare solo da due parti.

Quando si hanno tre o più possibilità, la tabella dei salti diventa un effettivo strumento di programmazione.

Il cuore di una tabella dei salti sarà una sequenza di indirizzi a 16 bit:

Supporremo che questi indirizzi di memoria contigui rappresentino gli indirizzi di partenza di un numero di programmi differenti. Supponendo che il programma richiesto sia identificato dal numero di programma nell'Accumulatore, **la seguente sequenza di istruzioni fa sì che l'esecuzione salti al programma il cui numero è immagazzinato nell'Accumulatore:**

```
; Salto al programma della tabella
        LD   HL,JTBL    ; Carica in HL l'indirizzo di base della tabella dei salti
        ADD  A          ; Moltiplica il programma # per due e
        LD   E,A        ; sposta il risultato nel registro e
        LD   D,0        ; posiziona a zero il registro D
        ADD  HL,DE      ; Addiziona il programma # volte a JBTL
        LD   E,(HL)     ; Carica in E il byte di indirizzo di ordine minore
        INC  HL         ; Incrementa il puntatore in HL
        LD   D,(HL)     ; Carica in D il byte di ordine maggiore dell'indirizzo
        EX   DE,HL      ; Metti in HL l'indirizzo per far partire il programma
        JP   (HL)       ; Salta all'inizio del programma
```

34

Programmazione dello Z80 e progettazione logica

Adam OSBORNE
Jerry KANE
Russell RECTOR
Susanna JACOBSON

GRUPPO EDITORIALE JACKSON